Power-balanced instruction scheduling for pipelined VLIW architectures

Xiao, Shu

2006

Xiao, S. (2006). Power-balanced instruction scheduling for pipelined VLIW architectures.
Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2483>

<https://doi.org/10.32657/10356/2483>

Nanyang Technological University

Downloaded on 26 Apr 2023 14:20:39 SGT

Power-Balanced Instruction Scheduling for Pipelined VLIW Architectures

Xiao Shu

School of Computer Engineering

A THESIS SUBMITTED TO
THE NANYANG TECHNOLOGICAL UNIVERSITY
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

2006

Abstract

Very Long Instruction Word (VLIW) processors are capable of executing a number of instructions which are scheduled by the compiler on their independent functional units in parallel. The power consumed at any given time by the processor depends on the number as well as the type of instructions that are being executed. Thus potentially large power fluctuations may occur throughout the duration of program, depending on how the instructions are scheduled by the compiler. Large processor current variation can accelerate battery exhaustion, increase power supply noise and degrade chip reliability. Furthermore, certain cryptographic applications also require relatively uniform processor power consumption to avoid security attacks based on observations of power behavior. The focus of this thesis is on techniques for minimizing power variation for the duration of the whole program. The problem of instruction scheduling for minimal processor power variation while the program execution is not degraded will be referred to as *power-balanced scheduling* in this thesis.

The optimization problem is first formulated as a mixed-integer program (MIP). While this approach has been proposed before, the original contributions made in this thesis are two-fold. Firstly, an efficient branch-and-bound algorithm is developed specifically for this problem enabling it to be solved much more efficiently than generic techniques. Secondly, a much more complete resource usage model is incorporated. This model takes into account the constraints imposed by multi-cycle functional unit latency as well as shared resources such as limited registers, read/write ports and buses which

have not been considered before. Experimental results based on Texas Instruments' TMS320C6711 VLIW digital signal processor using benchmark programs from MediaBench and Trimaran showed that over 40% average reduction in power variation can be achieved.

The quality of the optimized schedule obtained is dependent on the accuracy of the instruction-level power model of the target VLIW processor. The accuracy of the power model is affected by the precision of the power/current measurements and by the simplifications and assumptions made in order to cope with the complexity of the VLIW architecture such as inter-instruction, intra-instruction effects. It is desirable to take into account the imprecision of the power model so that the optimal power variation can be guaranteed to a certain degree of confidence. A new approach based on rough set theory is proposed to characterize the power model imprecision. The power model coefficients are represented as rough variables defined by their lower and upper approximations.

Based on the proposed rough power model, the optimization problem is reformulated as a chance-constrained rough program. The resulting optimal schedule guarantees to produce minimal power variation with the desired level of confidence. A new rule-based approach is developed to rank the candidate schedules efficiently during searching for the optimal schedules. A problem-specific genetic algorithm (GA) and branch-and-bound algorithm using the proposed rule-based approach are separately developed. While the GA can locate good schedules rapidly, the branch-and-bound algorithm guarantees optimality. The theory associated with these methods are mathematically justified.

Possible deviations from the optimal values due to the power model imprecision are reduced by over 20% and 30% on the MediaBench and Trimaran benchmarks respectively compared with the MIP solutions. The computational requirement of the rule-based algorithms is also shown to be low enough for this method to be practical for production compilers of embedded VLIW systems.

Acknowledgments

I would like to express sincere thanks to Dr. Edmund Lai, my supervisor, for his invaluable guidance throughout my Ph.D. research. *Give a man a fish and you will feed him for a day. Teach him how to fish and you will feed him for a lifetime.* I am indebted to him – a mentor who has been helping me all along, for his invaluable advice which will surely benefit me in my future.

I am grateful to Dr. Arlene Bastion for her professional and enthusiastic help in revising my thesis. Her constructive criticism has helped to improve this thesis substantially.

Finally, I would like to say a big thank you to Mr. Gao Shiyu. From him I learned to be optimistic and to positively confront the difficulties in my life.

Contents

Abstract	i
Acknowledgments	iii
List of Figures	ix
1 Introduction	1
1.1 Motivations	1
1.2 Related Works	2
1.2.1 Power-Balanced VLIW Instruction Scheduling	2
1.2.2 VLIW Instruction-level Power Models	4
1.3 Objectives and Contributions	6
1.3.1 Project Objectives	6
1.3.2 Original Contributions	7
1.4 Thesis Outline	9
2 Review of Instruction-level Power Models and Instruction Scheduling for VLIW Processors	10

2.1	VLIW Architecture	10
2.1.1	History of Development	11
2.1.2	A Commercial VLIW Processor	12
2.2	VLIW Instruction-level Power Models	13
2.2.1	An Instruction-Based Model	14
2.2.2	An Algorithmic Activity Based Model	19
2.2.3	Model Comparison and Evaluation	25
2.3	VLIW Instruction Scheduling	26
2.3.1	Schedule Constraints	26
2.3.2	Scheduling for Performance	28
2.3.3	Scheduling for Low Energy	31
2.3.4	Power-Balanced Instruction Scheduling	32
3	An Improved Power Model	36
3.1	Functional Blocks with Concurrent Activities	38
3.2	Instruction-Level Algorithmic Parameters	38
3.2.1	Power Model Coefficients	39
3.3	Power Consumption as a Function of Time	39
3.4	Power Model of the C6711	41
3.5	Summary	43

4	The Mixed-Integer Programming Approach	44
4.1	Problem Formulation	45
4.1.1	Notations	45
4.1.2	Mixed-Integer Program Formulation	47
4.2	A Branch-and-Bound Algorithm	55
4.2.1	Preliminaries	55
4.2.2	Branching	56
4.2.3	Selection	60
4.2.4	Lower Bound Estimation	60
4.3	Performance Evaluation	63
4.4	Summary	70
5	Basic Concepts of Rough Sets, Rough Variables and Rough Program	71
5.1	Rough Sets	71
5.2	Rough Variables	73
5.2.1	Discretization	73
5.2.2	Basic Rough Arithmetic	75
5.2.3	Ordering Rough Values	76
5.2.4	Rough Simulation	77
5.3	Rough Program	78

6	Instruction-level Power Model Using Rough Variables	82
6.1	Collecting Attribute Data	83
6.2	Discretization	83
6.3	Power Model Coefficients Represented by Rough Variables	84
6.4	Rough Power Model of C6711	84
6.5	Summary	87
7	The Rough Programming Method	88
7.1	Chance-Constrained Rough Program	89
7.1.1	The Objective function	90
7.1.2	Constraints	92
7.1.3	Solution	92
7.2	A Genetic Algorithm Using Rough Simulation to Rank Schedules	92
7.2.1	Chromosome encoding and initial population	93
7.2.2	Fitness evaluation	93
7.2.3	Genetic operations	93
7.3	A Rule-based Approach to Ranking Schedules	95
7.3.1	Theoretical Basis	95
7.3.2	Examples	99
7.4	A Genetic Algorithm with Rule-Based Schedule Ranking	101

7.5	Branch-and-Bound Algorithm with Rule-Based Schedule Ranking	102
7.5.1	Preliminaries	102
7.5.2	Branching Rules	102
7.5.3	Lower Bound Estimation	103
7.6	Performance Evaluation	107
7.7	Summary	111
8	Conclusions	113
8.1	Recommendations for Further Research	115
A	Effects of the number of parallel instructions on Power Consumption for the C6711	118
B	Scheduling for performance and for total energy minimization on C6711	121
C	Assembly code structure of C6711	125
	Bibliography	127

List of Figures

2.1	A general VLIW processor core [1].	11
2.2	Internal Organization of the TMS320C6711 [2].	13
2.3	Experimental platform for instruction-level power measurement used in [3].	16
2.4	Functional blocks with concurrent activities (adapted from [4]).	20
2.5	Example of an TMS320C6711 elementary program for activating only the IMU.	22
2.6	Example TMS320C6711 elementary program that activates the IMU and PU.	23
3.1	The improved power model (c.f. Figure 2.4).	37
3.2	Experimental setup for measuring processor current of TMS320C6711. . .	42
3.3	Example elementary program that activates the PU, IMU and DMU. . .	42
4.1	Convergence process of a problem instance with a size of (60, 39).	66
7.1	Data dependency graph of instructions in Example 4.	108
B.1	An C6711 assembly program which repeats two parallel “ADD”s.	121

B.2	An C6711 assembly program which repeats two parallel “MPY”s.	122
B.3	An C6711 assembly program which repeats four parallel instructions: two “ADD”s and two “MPY”s.	122
B.4	An C6711 assembly program which repeats two parallel “ADD”s and two parallel “MPY”s successively.	123

Chapter 1

Introduction

1.1 Motivations

Very Long Instruction Word (VLIW) processors are capable of executing several instructions in parallel on separate functional units. These processors are very suitable for algorithms for multimedia signal processing which typically exhibit a high degree of instruction-level parallelism (ILP) [5–7]. How well the processor exploits the ILP inherent in the application program is determined by the quality of the compiler [8].

The average power P consumed by a microprocessor while running a program is given by $P = I_{DD}V_{DD}$, where I_{DD} is the average current and V_{DD} is the supply voltage. Assuming that the supply voltage V_{DD} is fixed, the power cost is proportional to the current drawn by the CPU. The schedule of VLIW instructions produced by the compiler consists of a variable number of instructions being executed in parallel at any given time. The number of instructions may vary substantially from clock cycle to clock cycle. Hence there will potentially be large fluctuations in processor supply current because the power consumed at any given time in the processor depends on the number as well as the type of instructions that are being executed [1].

A typical example of a VLIW processor is the Texas Instruments' TMS320C6711 digi-

tal signal processor. It has an 8-issue VLIW architecture which means that it can execute up to 8 instructions in parallel. The impact of the number of parallel instructions on the power consumption of the CPU and internal memory, as demonstrated in Appendix A, can be quite substantial.

Large differences in the average current between consecutive clock cycles result in power supply voltage noise, which may cause timing and logic errors. For high-performance processors, this problem is becoming more serious because the increasing clock frequency, the growing number of gates, and the wider datapath result in larger surge current in a shorter time [9, 10]. Furthermore, large current variation is usually correlated with a large number of high peak current, which adversely affect the chip temperature [11] that the chip reliability and sub-threshold leakage power are exponentially related to [12]. Besides, the battery efficiency, for the same mean value of discharge current, may change by as much as 25% as a result of the discharge current profile over time. In particular, the maximum battery life is achieved when the variance of the discharge current distribution is minimized [13].

Thus there is a need to produce VLIW instruction schedules so that power fluctuations are minimized. Instruction scheduling for minimal power variation will be referred to as *power-balanced scheduling* in this thesis. We shall concentrate on techniques that will not degrade the speed performance of the programs.

1.2 Related Works

1.2.1 Power-Balanced VLIW Instruction Scheduling

Previously published works on power-balanced VLIW instruction scheduling are relatively scarce. There are two main approaches to the problem. The first approach is to extend the performance-oriented iterative modulo scheduling algorithm by adding

heuristics for minimizing processor power variation [14]. The alternative approach is to formulate it as a mixed-integer program (MIP) [15]. Although algorithms for solving the MIP generally have higher computational complexities than heuristic ones, they can provide us with optimal instruction schedules while heuristic solutions cannot guarantee optimality. Some extra computational effort can be justified by the fact that, for embedded systems, the application is compiled only once at the vendor site but it is executed many times at the customer site. Hence code quality is more important than compilation time.

However, the time complexity is particularly high if the program is solved using generic MIP solvers, as is done in [15]. Furthermore, these previous works assume a very simplistic resource usage model for their target VLIW processors. First, they assume that the instructions scheduled to each functional unit were executed on dedicated resources in a fully pipelined manner. In fact, instructions with multi-cycle functional unit latency will lock the target functional unit for the necessary number of cycles. No new instruction can be dispatched to that functional unit during this locking period. Second, they do not consider the issue of register utilization. Rescheduling instructions can change variables' lifetimes, which may increase pressure on the registers. Without enough registers, the register allocator must insert spill and restore code in the schedule. This may cause extra delay and increase the total energy consumption as well as power variation of the resulting schedule. Third, they do not take into account the sharing of resources, such as read/write ports and shared buses, between the functional units. As a result, illegal schedules may be obtained.

The instruction-level power models used in [14, 15] are also rather simplistic. In [14], a power cost is associated with each individual instruction, independent of other instructions. The inter-instruction and intra-instruction effects, which are significant in VLIW architectures [1, 3, 16], are not modelled at all. The simplified power model used

has not been validated. So we have no information on its accuracy. The power model used in [15], on the other hand, assumes that all pipeline stages in each functional unit consume equal power. The total amount of consumed power is therefore computed as the sum of power consumed by the active pipeline stages of all the functional units. As shown in the experimentally verified VLIW instruction-level power model proposed in [4], significant differences exist between the power consumed by the different stages. Hence the optimal schedules obtained based on these rather simplistic power models may be far from optimal in reality.

1.2.2 VLIW Instruction-level Power Models

Two instruction-level power models specifically for VLIW architectures can be found in the literature. The first one, presented in [1, 17–20], is very accurate, with an average error of 4.8% compared to gate-level simulations. It takes into account factors such as instruction ordering and power consumption in the pipeline stage. However, due to its complexity, a large number of parameters have to be estimated. Since these parameters are normally obtained via power measurements, the number of measurements to be made is prohibitively large. If this complex model is applied to instruction scheduling, the computational efficiency of the scheduling algorithm will be affected.

The second power model, described in [4], has a lower complexity. The power cost is estimated through the parameters which characterize the instruction-level algorithmic activity of the target program. The architectural components in the target VLIW processor are grouped based on a functional-level concurrent activity analysis. The instruction-level algorithmic parameters actually represent the activity rate of the component groups and their interactions. The underlying idea is based on the observation that the architectural complexity of VLIW processors hides many activity variations. Hence no significant difference can be observed between certain types of operations. For instance, for the

TMS320C6201 digital signal processor, an addition and a multiplication dissipate about the same amount of power. The same is true for a data transfer with the internal memory. Although the complexity is lower, the accuracy of this model is comparable with the previous one. The maximum errors between estimations and measurements are 4% for the C62 and 6% for the C67 series of digital signal processors. Furthermore, the methodology can be applied to other VLIW processors.

There are also some shortcomings of this model as reported in [4]. The functional-level concurrent analysis assumes that the internal data memory and the processing units always have concurrent activities. This is not true. There are concurrent activities only when the instructions being executed require data memory access. It is also assumed that when the pipeline executes a NOP (no-operation) instruction, it is still dispatched to a functional unit. In fact, when the pipeline executes a NOP, it is not dispatched to a functional unit because there is no execution associated with it. These inaccuracies in the functional-level concurrent analysis reduce the accuracy of the resulting instruction-level power model. This model also does not provide any insight into the overlapping of instructions within the pipeline. Hence we cannot use it to provide a good run-time instruction-level power profile for the software being executed. If this model were to be useful to our power-balanced instruction scheduling problem, it will have to be modified to allow us to compute the power consumption as a function of time.

Regardless of the power model used, its parameters will need to be estimated. It can be done either by cycle-level simulations or by physical measurements. Simulation based tools such as Simple-Power [21] and Wattch [22] require details of the microarchitecture of the processor. These details are often unavailable for commercial off-the-shelf processors. Even if they are available, the computational costs of these simulations for VLIW processors are prohibitively high.

Measurement based methods rely on empirical measurements of the current drawn by

the processor during repeated execution of individual instructions [3, 4, 16]. While it is more efficient, there will always be a certain degree of imprecision involved in measurements. The standard practice is therefore to perform repeated experiments and use the mean or median values in the power model. Based on such models, the optimal schedules obtained can only be power balanced in an average sense. For example, one cannot guarantee that the optimal schedule obtained with such power models that a hard power variation limit for chip reliability will not be exceeded in practice.

1.3 Objectives and Contributions

1.3.1 Project Objectives

The first objective of this research is to develop instruction scheduling techniques for VLIW processors to achieve minimal power variation throughout the duration of the program. In order to ensure that the execution performance of the target program is not compromised, we can start with a performance optimal schedule produced by a performance-oriented VLIW instruction scheduling algorithm [8, 23]. Then the power-balanced schedule can be obtained through rescheduling. In this way, the speed performance of the program would not be affected. Thus the techniques described in this thesis are in fact rescheduling algorithms.

In order to achieve this objective, the related issue of power modelling have to be tackled. Existing power models for VLIW processors are not suitable for our needs due to the reasons presented in Section 1.2.2. They need to be modified to suit our needs for instruction scheduling. The modified models must be computationally efficient and yet detailed enough to provide a power profile of the target program cycle by cycle.

For power critical applications, it is important to guarantee that the power variation of the program stay within the designed tolerance with a prescribed level of confidence. The

second objective of this project is to come up with power-balanced instruction scheduling algorithms that can provide this kind of guarantee. Naturally this also means that the power model should be accurate to desired level of confidence.

1.3.2 Original Contributions

Both objectives set out for this project have been achieved. The first approach, which uses a mixed-integer programming formulation together with a customized branch-and-bound algorithm, provides a solution for the first problem. The approach to the second problem is through the relatively new rough set theory and rough programming.

The following original contributions are related to the MIP approach.

1. The power model for VLIW processors in [4] has been modified for the instruction scheduling problem. A new grouping strategy that is based on a more precise functional concurrent activity analysis is used. As a result, the accuracy of the original model is enhanced.

The original model only allows us to compute the total power consumed. By accounting for the overlapping of instructions within the pipeline for each group of architectural components, it has been adapted so that fine-grained run-time power profiling can be performed.

2. A much more complete resource usage model than those used in previous works is adopted. This model takes into account constraints imposed by multi-cycle functional unit latency as well as limited registers and shared resources such as read/write ports and buses which have not been considered before. The quality of the solutions is improved as a result.
3. An efficient branch-and-bound algorithm is developed to solve the MIP more efficiently than generic solvers, making the technique more attractive for use in prac-

tical compilers. Generic MIP solvers do not have problem-specific knowledge to reduce the search space and therefore are inferior to our branch-and-bound algorithm which incorporates such knowledge specifically for this problem so that optimal solutions can be found efficiently.

The original contributions related to the rough programming approach are as follows.

1. A novel instruction-level power model with model coefficients represented as rough variables is introduced, which characterize the imprecision due to both power measurements and the simplifications made to the VLIW architecture.
2. Given the rough power model of the target VLIW processor, a new formulation of the power-balanced instruction scheduling problem is proposed. The optimization problem is reformulated as a chance-constrained rough program. The resulting optimal schedule is guaranteed to produce minimal power variation with a specified level of confidence.
3. In the process of searching for the optimal schedule, ranking and ordering of candidate schedules based on their objective function values have to be performed. Since the objective function values are themselves rough values, a process which involves rough simulation [24] is required. Rough simulation is very computationally expensive. A new rule-based approach is proposed to rank the candidate schedules very efficiently. The theory associated with this method is formally (mathematically) established.
4. A problem-specific genetic algorithm (GA) and a branch-and-bound algorithm have been developed to solve the rough program. While the GA cannot guarantee optimality, the branch-and-bound algorithm does, albeit with a higher time complexity.

The efficiency and effectiveness of these algorithms are confirmed using the C6711 [2]

VLIW digital signal processor. The multimedia benchmark programs used are obtained from MediaBench [25] and Trimaran [26] which are generally accepted by the research community.

1.4 Thesis Outline

The rest of this thesis is organized as follows.

A review of the published VLIW instruction-level power models and VLIW instruction scheduling methods is given in Chapter 2. Our extended and modified power model is presented in Chapter 3. Our mixed-integer program problem formulation and the proposed branch-and-bound solution are covered in detail in Chapter 4.

Chapters 5 - 7 present the rough programming approach. Chapter 5 introduces the concepts of rough set, rough variables and rough programming. Chapter 6 describes the new VLIW instruction-level power model using rough variables. The proposed chance-constrained rough program problem formulation, the rule-based schedule ranking approach and the problem-specific GA and branch-and-bound algorithms are discussed in Chapter 7.

Finally, Chapter 8 concludes with a summary of the research and proposes possible future work.

Chapter 2

Review of Instruction-level Power Models and Instruction Scheduling for VLIW Processors

A good instruction-level power model for the target VLIW processor is very important for the power-balanced instruction scheduling problem. In this chapter, we shall first give a brief overview of the VLIW architecture, using the C67 series of digital signal processors as an example. Then a review of existing instruction level power models is given, followed by a discussion of previously published works on power-balanced instruction scheduling.

2.1 VLIW Architecture

A general VLIW architecture is illustrated in Figure 2.1. It has a number of functional units that can execute individual instructions in parallel. The term VLIW refers to the size of each instruction word that is carried out by a processor. This instruction word is “very long” in comparison to the instruction word size utilized by most current mainstream processors. The parallel architecture can achieve very high performance if the software has a high degree of instruction level parallelism (ILP). It also depends heavily on the compiler’s ability to exploit the ILP inherent in the program.

Figure 2.1: A general VLIW processor core [1].

A variant of VLIW is the Explicitly Parallel Instruction Computing (EPIC) architecture [27]. EPIC has a few additional ideas taken from superscalar processors. Basically, it makes use of VLIW instructions, but with some “hints” embedded in the instruction set to help with the parallelism and to reduce branching with predication and some other minor changes.

2.1.1 History of Development

VLIW processing ideas have roots in the microprogramming domain [28]. When Joseph Fisher was working on writing horizontal microcode in 1979, he began to work on the problem of generating long instruction words from short sequential instructions [29]. In 1984, Fisher and his colleagues at Yale University formed a company called Multiflow, with the goal of creating VLIW supercomputers [30,31]. At around the same time, the company Cydrome was formed by Rau [32,33] with similar aims. Although they both released finished products, the marketplace was not ready.

Little further development appeared on the VLIW front until the 1990's, when a new wave of media processing chips appeared. The first ones are Chromatic's MPact and Philips' TriMedia, followed by Texas Instruments' C6X series and Analog Device's TigerSHARC series of digital signal processors. The success of the C6X chips paved the way for a new generation of large-scale VLIW microprocessor designs. In 2000, Transmeta released their family of VLIW-based Crusoe processors, designed especially for low-power processing in embedded devices. Intel followed with their 64-bit architecture specification, the IA-64 [34], and continued with the Itanium processors with EPIC architectures.

2.1.2 A Commercial VLIW Processor

The Texas Instruments' C6x family of microprocessors is a popular, commercial off-the-shelf VLIW processor family. They are digital signal processor chips, built around Texas Instruments' VelociTI VLIW architecture. Digital signal processing applications are well-suited for a VLIW architecture since there is a great deal of inherent parallelism in the algorithms. Wideband modems (ADSL), real-time image processing, and wireless telecommunications are a few examples of the applications of this technology.

All the examples and experimental evaluation of algorithms in this thesis will be based on Texas Instruments' C6711 VLIW digital signal processor (DSP). Figure 2.2 illustrates the internal organization of the C6711. Its CPU contains a program fetch unit, an instruction dispatch unit, an instruction decode unit, two data paths, thirty-two 32-bit registers, control registers, as well as control, test, emulation, and interrupt logic. The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units in every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units denoted by ".L", ".S", ".M", and ".D". The internal (on-chip) memory is organized as separate data and program spaces. It has a two-level memory

Figure 2.2: Internal Organization of the TMS320C6711 [2].

architecture for program and data. The first-level program cache is designated $L1P$, and the first-level data cache is designated $L1D$. Both the program and data memory share the configurable second-level memory, designated $L2$. When off-chip memory is used, these spaces are unified on most devices to a single memory space via the external memory interface (EMIF).

2.2 VLIW Instruction-level Power Models

VLIW power modelling can be either cycle-level simulation-based or physical measurement based. The simulation based tools such as Simple-Power [21] and Wattch [22] require details of the microarchitecture, which are often unavailable for commercial off-the-shelf processors. Even if these details are available, the computational costs of these simulations for VLIW processors are prohibitively high. The measurement based method [3, 4, 16] relies on empirical measurements of the current drawn by the processor during repeated execution of individual instructions. While it is more efficient, the major disad-

vantage is that there will always be some degree of imprecision involved in measurements. For commercial off-the-shelf processors, we have no choice but to use the measurement based method.

Two power models have been proposed specifically for VLIW processors. The first one is a purely instruction based model while the second one models the power costs of the algorithmic activities in the functional units. They are described in detail in Sections 2.2.1 and 2.2.2 respectively.

2.2.1 An Instruction-Based Model

The method of instruction-level power/energy modelling based on physical measurements was first proposed in [16]. It estimates the power/energy cost of the program when there are no pipeline stalls and cache misses. The total power cost is expressed as the sum of the base cost of each executed instruction in the program and the cost of the inter-instruction effects.

Each instruction involves a number of units in the processor. This can result in circuit switching activities that are characteristic of each instruction and which vary from instruction to instruction. If a given instruction is executed repeatedly, then the power/energy consumed by the processor can be thought of as the power/energy cost of that instruction. This is the base cost of the instruction.

The circuit switching activity is a function of the present state and the previous state of the circuit. Since the base cost is determined by executing the same instruction repeatedly, the measurement is performed in a context of least change. However, when a pair of different instructions are executed sequentially, there are certain inter-instruction effects that has not been taken into account by the base cost. The difference is the cost of the inter-instruction effects. The cost of a pair of successively executed instructions is

always greater than the sum of base cost of each individual instruction.

Mathematically, the average energy can be expressed as

$$E = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{ij} \times N_{ij}) \quad (2.1)$$

where E is the total energy dissipation associated with the execution of a given program when there are no pipeline stalls and cache misses. The first term is the sum of the base energy cost (B_i) of each instruction i multiplied by the number of times (N_i) that the instruction is executed. The second term is the sum of the additional energy cost O_{ij} of the instruction sequence i followed by j multiplied by the number of times (N_{ij}) that the instruction pair is executed.

2.2.1.1 Experimental Setup

The power costs B_i and O_{ij} in (2.1) for a target processor are determined through measurements. The average power P consumed by a microprocessor while running a program is given by

$$P = I_{DD}V_{DD} \quad (2.2)$$

where I_{DD} is the average current and V_{DD} is the supply voltage. Thus, the current drawn by the CPU during the execution of the program is proportional to its power cost. The ability to measure the current drawn by the CPU during the execution of the program is essential for measuring its power cost.

In order to measure the processor current, the power supply connections to the CPU are isolated from the rest of the system. When executing the instruction sequences, the current drawn by the processor is measured through a standard off-the-shelf digital ammeter. An improvement of the current measurement method that uses a digitizing oscilloscope was proposed in [3]. Figure 2.3 shows the measurement configuration. To measure the current, a precision resistor was placed in series with the power connection

Figure 2.3: Experimental platform for instruction-level power measurement used in [3].

to the processor. This introduced stray inductance in the power delivery circuit, which in turn induces a noise voltage during switching surge currents. To counteract this effect, decoupling capacitors were added to the wires. Experiments with different capacitor values are tried to reduce the high frequency component of the noise.

The target instruction or instruction pair was placed in an infinite loop in order to obtain a stable current reading. The size of the loops has to be carefully considered. As more of the target instructions are put in the loop, the impact of the branching at the end of the loop is minimized. Very large loops, on the other hand, may cause cache misses, which are undesirable during determination of base costs. Only the target instructions should execute on the CPU during experiments; multi-tasking and interrupts are disabled. An average value can be obtained from the digitizing oscilloscope.

While this method seems intuitive if the CPU executes only one instruction at a time,

most modern CPUs process more than one instruction due to pipelining. However, what follows shows that the concept of a base energy cost per instruction remains unchanged with pipelining.

Suppose a CPU has a five-stage pipeline. Let $E_{j_{INS_k}}$ be the average energy consumed by pipeline stage j , when instruction INS_k executes in that stage. Pipeline stages are separated from each other by latches. Thus, if the effect of circuit state and resource constraints are ignored for now, the energy consumption of different stages is independent of each other. Suppose that in a given cycle, instruction INS_1 is being processed by stage 1, INS_2 by stage 2, and so on. The total energy consumed by the CPU in that cycle would be

$$E_{cycle} = E_{1_{INS_1}} + E_{2_{INS_2}} + E_{3_{INS_3}} + E_{4_{INS_4}} + E_{5_{INS_5}} \quad (2.3)$$

On the other hand, the total energy consumed by a given instruction INS_1 , as it moves through the various stages is

$$E_{INS_1} = \sum_j E_{j_{INS_1}} \quad (2.4)$$

This quantity actually refers to the base cost as described above. This method of forming a loop of instances of instruction INS_1 , results in $E_{cycle} = E_{INS_1}$, since in that case, $INS_1 = INS_2 = INS_3 = INS_4 = INS_5$. The average current in this case is $\sum_j E_{j_{INS_1}} / (V_{DD} \times \tau)$, which is the same as the digital oscilloscope reading obtained, where τ denotes the clock period.

2.2.1.2 Model Simplifications

The instruction-level power model (2.1) must characterize the base cost of each executed instruction and the inter-instruction effect of any successive instruction pair. Let N be the number of instructions in the instruction set of target VLIW processor and K be the long instruction word width. Assume that a long instruction word may consist of 1 to K arbitrary combination of instructions. Then the number of long instruction words that

can be created is N^K . This means that the number of parameters B_i that have to be characterized is N^K and the number of O_{ij} is N^{2K} . These numbers are large even for modest values of K and N .

The spatial additive property was introduced in [1, 19, 20] to reduce the complexity of the power model. The energy/power is decomposed into the sum of the energy/power contributions due to each instruction involved in the execution of the long instruction words. It has been experimentally verified that mutual influence between two different instructions in the same long instruction word is insignificant compared to that between instructions in two successive long instructions words [1]. This is called the spatial additive property of the energy/power of a VLIW processor. By using this property, the complexity of the model can be reduced from $O(N^{2K})$ to $O(KN^2)$.

Let $v_{(i)}^k$ denote the base energy cost consumed by an instruction executed on functional unit k at the i -th time slot and let $v_{(i|i-1)}^k$ denote the additional energy contribution due to inter-instruction effect. Given a VLIW instruction schedule with a total of t time slots, the energy consumed during the execution of this schedule using the reduced complexity model can be expressed as

$$E = \sum_{i=1}^t \sum_{k=1}^K (v_{(i)}^k + v_{(i|i-1)}^k) \quad (2.5)$$

The number of instruction pairs $v_{(i|i-1)}^k$ in (2.5) that have to be considered is still prohibitively large. This is because we have to consider instructions that differ in functionality (i.e., opcode), addressing mode (immediate, register, indirect, etc.) as well as data differences either in terms of register names or immediate values. The complexity of model can be reduced by grouping the instructions with similar power costs into a cluster [18].

A number of clustering algorithms can be used [35]. The C -means clustering algorithm was chosen in [18] since it has the lowest computational complexity. Let C be the number

of instruction clusters. The C -means clustering algorithm partitions them into C clusters ($c_1 \dots c_C$) by minimizing the mean-squared error

$$\sum_{j=1}^C \sum_{i=1}^{n_j} (x_{ij} - g_j)^2$$

where n_j denotes the number of elements in cluster c_j , x_{ij} is the i -th element in cluster c_j . g_j denotes the center of gravity of the cluster c_j .

In the power model, instructions in the same cluster can share the same power coefficient. Hence (2.5) becomes

$$E = \sum_{i=1}^t \sum_{k=1}^C c_k r_k^i \quad (2.6)$$

where c_k is the center of gravity of the k -th cluster and r_k^i denotes the number of instructions being executed at time slot i which belong to cluster k . The model complexity is now reduced from $O(KN^2)$ to $O(KC^2)$ after instruction clustering.

2.2.2 An Algorithmic Activity Based Model

An alternative VLIW instruction-level power model was proposed in [4]. In this model, the power parameters characterize the instruction-level algorithmic activity of the target program. The architectural components in the target VLIW processor are grouped. Grouping is based on a functional-level concurrent activity analysis. The instruction-level algorithmic parameters represent the activity rate of the groups and their interactions. This power modelling methodology can be applied to other VLIW processors although it was only illustrated using the C6201 digital signal processor in [4].

2.2.2.1 Functional Blocks with Concurrent Activities

The architecture of the target VLIW processor is represented in this model by two main functional blocks as shown in Figure 2.4. They are the instructions management unit

Figure 2.4: Functional blocks with concurrent activities (adapted from [4]).

(IMU) and the processing unit (PU). They model the concurrent activities in the pipeline steps when the processor operates.

The pipeline steps can be grouped into three stages:

1. The fetch stage. It includes program address generation (PG), program address send (PS), program access ready wait (PW) and program fetch packet receive (PR), which fetch the instructions from program memory.
2. The decode stage. The instructions are first dispatched to the correct unit (DP) and then decoded by the receiving processing unit (DC).
3. The execution stage. The instruction is executed in a variable number of steps.

As shown in Figure 2.4, the four steps in the fetch stage and the instruction dispatch (DP) step in the decode stage activate the IMU. The instruction decode (DC) step in the decode stage and the execute stage activate the PU.

2.2.2.2 Instruction-Level Algorithmic Parameters

The activity rate of the functional blocks and their interactions are modelled by two algorithmic parameters. These parameters are obtained from the compiled code and have a significant impact on the final power consumption. The parallelism rate α indicates the relative flow between the fetch stages and the program memory controller in the IMU. The processing rate β between the IMU and the PU represents the utilization rate of the processing units. α and β are computed by

$$\alpha = NFP/NEP \quad (2.7)$$

$$\beta = \frac{1}{NEP} \left(\frac{NPU}{NPU_{\max}} \right) \quad (2.8)$$

```

LOOP :
    NOP
    NOP
    ...
    NOP
    B    .S1  LOOP

```

Figure 2.5: Example of an TMS320C6711 elementary program for activating only the IMU.

where NFP and NEP are the average number of fetch packets and execution packets respectively. Thus $\alpha \leq 1$ with the maximum $\alpha = 1$ attained when parallelism is highest. NPU is the average number of processing units used per cycle and NPU_{max} is the maximum number of processing units used per cycle. Hence, $\beta \leq \alpha$.

2.2.2.3 Power Model Coefficients

The total power I_{tot} consumed by the processor when there is no instruction or data cache misses is reflected by the power consumed by the IMU and the PU.

$$I_{tot} = I_{IMU} + I_{PU} + e \quad (2.9)$$

where

$$I_{IMU} = a\alpha \quad (2.10)$$

$$I_{PU} = b\beta \quad (2.11)$$

and e is the constant coefficient. Its physical meaning is the power consumed by the processor which is powered on but does not fetch or execute any instructions.

The supply current to the processor I_{tot} is measured in order to estimate the values of the power model coefficients a , b , and e in (2.9) to (2.11). The measurement setup is similar to those in [3, 16]. A set of *elementary programs* with various instruction-level algorithmic activities are designed to activate the functional blocks (IMU and PU). In

```

LOOP :
    ADD .L1 A1, A2, A3
||     ADD .L2 B1, B2, B3
    ADD .L1 A1, A2, A3
||     ADD .L2 B1, B2, B3
    ...   ...
    ADD .L1 A1, A2, A3
||     ADD .L2 B1, B2, B3
    B     .S1     LOOP

```

Figure 2.6: Example TMS320C6711 elementary program that activates the IMU and PU.

such an elementary program a long instruction word is repeated by explicitly following it with an unconditional branch back to the top of it. The long instruction word is placed in such an infinite loop in order to obtain a stable current reading. Take TI TMS320C6711 [2] as an example, in the elementary program shown in Figure 2.5, only the IMU is activated when executing NOPs. One more example program are shown in Figures 2.6. The program in Figure 2.6 activates both the PU and IMU when executing “ADD”. Appendix C contains an explanation of the assembly code structure of the C6711.

The loops have to satisfy certain size constraints. On one side, the more the target long instruction words are put in the loop, the less the impact of the branch at the end of the loop. On the other hand, the loop size should not be too large that cache misses are caused, which are undesirable during determination of the power model coefficients a , b and e .

An elementary program q provides the values of its parameters α and β and the measured processor’s average current I_{tot} while executing this elementary program q . Suppose N elementary program experiments are conducted. Then the power model coefficients can be estimated by linear regression that minimizes the sum of squares of the residuals $\sum_{q=1}^N (a\alpha + b\beta + e - I_{tot})^2$. In order to cope with the effect of nuisance factors

and the imprecision inherent in measurements, it is necessary to produce a large number of elementary program experiments N for the linear regression.

This power modelling methodology can be applied to other VLIW processors although it was only illustrated using the C6201 digital signal processor in [4]. The C67 is a floating-point processor with an architecture close to that of the C62. These two DSPs therefore have the same functional block analysis and instruction-level algorithmic parameters for the processor models but different coefficient values. The maximum error between the estimates and the physical measurements of the C67 was 6 percent [4].

The power consumption laws that form the power model are therefore able to take into account all the power consumption sources including pipelines and internal memories. If pipeline stalls are considered, the effective α and β are given by α' and β' respectively.

$$\alpha' = \alpha(1 - PSR)$$

$$\beta' = \beta(1 - PSR)$$

where PSR is the pipeline stall rate. If DMA is used to transfer data to the internal memory, the power consumed by DMA should also be taken into account. In this case, (2.9) becomes

$$I_{tot} = I_{IMU} + I_{PU} + I_{DMA} + e \quad (2.12)$$

with

$$I_{DMA} = d\varepsilon \quad (2.13)$$

The direct memory access (DMA) utilization rate ε represents the activity level between the data memory controller and the DMA. ε has to be estimated by the programmer.

The power-balanced VLIW instruction scheduling techniques discussed in this thesis are only for VLIW instruction scheduling at the compile time for minimum power variation of the target processor. However, the power variation during the execution of the

program would be affected by instruction and data cache misses which are not considered in this thesis.

2.2.3 Model Comparison and Evaluation

The power model (2.5) is very accurate, with an average error of 4.8% compared to gate-level simulations [1]. However due to inter-instruction effects, a large number of parameters have to be estimated. Since these parameters are normally obtained via power measurements, the number of measurements that needs to be made is prohibitively large. Furthermore, it also reduces the computational efficiency of the instruction scheduling algorithm.

Compared with (2.5), the power model (2.6) has lower complexity. However, the power coefficients c_k in (2.6) are inexact due to the clustering of instructions described in Section 2.2.1.2 as well as imprecision of power measurements. In Chapter 6, we shall propose a method based on rough set theory to take the inexactness into account.

The model given by (2.9) has the lowest complexity compared with the previous two models, (2.5) and (2.6). The underlying idea is based on the observation that the architectural complexity of VLIW processors hides many activity variations. A major part of the circuit activity in a VLIW processor is common to all instructions, e.g. instruction pre-fetch, pipeline control, clock etc. While there may be significant activity variation within certain modules, its impact on the overall energy cost is small compared to the common cost. For instance, for the C6201 digital signal processor, an addition and a multiplication dissipate about the same amount of power. The same is true for a read or write to internal memory. Further, the experiments with pairs of instructions of Intel's 486DX2 reveal that the inter-instruction effect has a limited range—between 5.0mA and 30.0mA and most frequently occurred in the vicinity of 15.0mA [16]. The limited variation in inter-instruction effects is also true for VLIW processors. Although

the complexity of model (2.9) is lower, the accuracy is comparable with (2.5). The largest errors between estimations and measurements are 4% for the C62 and 6% for the C67 series digital signal processors [4].

The power model that is used in our mixed-integer programming approach to instruction scheduling is based on the power model (2.9). However, some modifications are necessary. These modifications are discussed in Chapter 3.

2.3 VLIW Instruction Scheduling

The effectiveness of a VLIW process depends very much on how the compiler schedules parallel instructions. The goal of any scheduling algorithm is to minimize a cost function while maintaining the semantics of the program and obeying the resource limitations of the target hardware. The cost function may reflect the speed, memory or power requirements.

In this section, we shall provide a overview of the process of instruction scheduling for VLIW architectures in general. This is followed by a review of previously published works on power-balanced scheduling.

2.3.1 Schedule Constraints

2.3.1.1 Data Dependency

Any instruction schedule must obey the data dependencies of the target program. There are three types of data dependencies – read-after-write, write-after-read and write-after-write. Read-after-write (RAW) dependency is also known as flow or true dependency. It occurs when one instruction uses the result of another. Hence a reordering of instructions would break this flow of data in the program. Write-after-read (WAR) dependency, also

called anti-dependency, occurs when one instruction overwrites a value after it has been used by another instruction. In this case, reordering would overwrite the correct value before it is used. Write-after-write (WAW) dependency (or output dependency) occurs when two instructions write to the same location. A reordering of the instructions will result in a wrong final value. The latter two kinds of dependencies, WAR and WAW, are also called false dependencies because they can be removed by renaming, i.e. adding extra temporary variables to the program.

Data dependencies are constraints that render some instruction reordering illegal. These constraints induce a partial ordering on the instructions. Any partial ordering can be represented as a directed acyclic graph (DAG) with nodes representing instructions and arcs representing the data-dependence constraints. The DAG represents the constraints that the scheduler must obey to maintain program semantics. But it also includes information that allows the scheduler to evaluate the relative order and importance of the instructions.

For a given DAG, the depth d_i of each node i is the length of the longest path from any root node. Similarly, the height h_i of node i is the length of the longest path to any leaf of the DAG. Let h_{max} be the maximum height. Then those instructions (nodes) for which $d_i = h_{max} - h_i$ are on the critical path of the region. For noncritical instructions, the range $[d_i, h_{max} - h_i]$ are the time slots in the schedule where the instruction can be placed without increasing the schedule length.

2.3.1.2 Resource Management

In addition to data dependencies, resource management is the other major constraint during scheduling. While dependencies and operational latencies may allow an instruction to be scheduled in a particular clock cycle, the target machine may not have sufficient number of suitable functional units, issue slots, or other hardware to launch or execute

the instruction at that time. These constraints on scheduling are called resource hazards, and a separate module of the scheduler typically models all of them. This module maintains its own state in response to scheduling actions, and answers queries about whether a given instruction can be scheduled in a particular place given the instructions already scheduled.

2.3.2 Scheduling for Performance

In order to achieve maximum performance, the scheduler should make as many functional units as busy as possible in each clock cycle while the data dependency and resource constraints are satisfied. This scheduling problem is equivalent to the “job shop scheduling” problem [36] which is NP-complete. However, simple heuristics applied to a very limited region of code generally work in practice. This is referred to as “local scheduling”. The small region of code, called a “basic block”, has no branches out except at the end of the block and no branches in except at the top. List scheduling [37] is a popular local scheduling technique.

2.3.2.1 List Scheduling

Most of the scheduling algorithms that operate on DAGs fall into the category of list scheduling. List scheduling techniques work by repeatedly assigning a cycle to an instruction without backtracking (greedy algorithms). Efficient implementations exhibit a computational complexity of $N \log(N)$, in addition to the DAG creation, which has N^2 complexity.

List scheduling selects an instruction from a data-ready queue (DRQ) of instructions. An instruction is ready when all of its DAG predecessors have been scheduled. Once scheduled, the instruction is removed from the DRQ and its successors that have become ready are inserted. This process is repeated until all instructions in the region are sched-

uled. The performance of list scheduling is highly dependent on the order used to select the scheduling candidates from the DRQ and, in the case of cycle-based scheduling, on the scheduler's greediness.

The first problem (DRQ order) is tackled by assigning a priority function to each instruction in the DAG. Ideally, the height of the instruction is what should drive the priority function since at any point during scheduling, instructions with the greatest height are the most critical. However, depending on other scheduling considerations, the depth or a combination of depth and height or a depth-first topological sorting of instruction within a connected component of a region are also used. In general, all these choices are based on heuristics, and have to strike a balance among schedule quality, implementation complexity and compile-time performance.

The second problem (greed control) is more complex since cycle-based schedulers often tend to be too greedy. Greediness may hurt performance in two ways. Firstly, instructions that are scheduled too early and that occupy resources for multiple cycles may prevent more critical instructions that become ready later to be scheduled due to resource constraints. Secondly, instructions that are scheduled too early may unnecessarily increase register pressure and force spills that could be avoided if they had been decided to delay their schedule.

2.3.2.2 Instruction Scheduling on Regions

One would not want to move an operation before a branch when there was a chance that control might transfer to the wrong leg of the branch. Thus intuitively it did not make sense to schedule across branches. But as these techniques were proposed and investigated, researchers noticed that too many "slots" remained unused in scheduling instructions for VLIW processors, i.e. hand rearrangements still seemed to use more of what the hardware was capable of. This shortfall occurred because instructions from

other basic blocks could fill the slots in ways that considered the branch problem and accounted for it. For example, one might shorten a schedule by moving an instruction down past a branch, making a copy of the instruction in both basic blocks that were targets of the branch. Sometimes this could eliminate a cycle in the earlier block without adding cycles to one or both of the target blocks.

Experiment had indicated that most of the opportunity for shortening schedules lie beyond branch boundaries [38]. Most research generally acknowledge that the boundary caused by branches was the largest limiting factor. This motivated a technique called *region scheduling* to enlarge the VLIW instruction scheduling scope. Here, “region” refers to subsections of a program that are compiled at the same time.

A particular region scheduling technique is trace scheduling. The basic idea is to select code originating in a large region, typically many basic blocks, before scheduling. Then instructions in the region are scheduled as if it were one big basic block. A region scheduling compiler typically follows this sequence in its scheduler.

1. Given the representation of the code in the compiler’s intermediate form, pick a region from the as-yet-unscheduled code. Typically, but not necessarily, a region is a set of contiguous basic blocks. It is simply a set of instructions originating in more than one basic block. Sometimes a code transformation is done prior to region selection, with the goal of enhancing the region selected. Enhancing could mean, for example, making the region larger.
2. Place the selected instructions on a data-precedence graph. Sometimes the edges or the instructions are decorated with additional information that is only important to region scheduling. In addition, some special edges may be added to the graph that prohibit illegal or undesirable code movements. These edges prevent code movements that would be illegal because of a violation in flow control that cannot

be compensated for when the schedule is produced. Such special edges are not required for local scheduling, since there is no flow control to consider within a basic block.

3. Schedule the instructions.
4. Either along the way, or in the final phase, do any necessary fix-ups. Usually these fixes take the form of extra instructions that mitigate what would have been an illegal transformations caused by the positions of the scheduled instructions. For example, sometimes an instruction is moved in such a way that it should have been executed along some path through the code, but is not. In that case, a copy of the instruction is placed in the path from which it is missing. These added instructions are often called compensation code.
5. Go back to step 1 until no unscheduled code remains.

Many variants of region scheduling algorithms have been proposed. But most follow essentially this framework. Typically, the algorithms vary in the way the region is selected in step 1 and in the complexity of the schedule construction pass in step 3. A survey of these techniques can be found in [8].

2.3.3 Scheduling for Low Energy

While scheduling for speed is most common, the total energy required to execute a software can also be the cost function to be minimized. The compiler of VLIW processors exploits the ILP and schedules a variable number of instructions to be executed in parallel on different functional units. Although this increases average power dissipation, the total number of cycles required to execute the program is reduced. The total energy consumed by the program, which is the product of average power dissipation and the total number

of cycles to execute the program, is actually less than the executions with less parallelism of these instructions on VLIW processors. In [4], the real physical measurement of power and energy consumption is conducted with a C6201 digital signal processor. Our measurement results with the C6711 (see Appendix B) also confirms this observation. Therefore there is no conflict between VLIW instruction scheduling for performance and for low energy. The scheduling techniques for performance optimizations are adaptable to total energy minimization. Apart from minimizing the energy consumption in the processor core, techniques to reduce energy consumption in the instruction bus are proposed in [39–41].

2.3.4 Power-Balanced Instruction Scheduling

Previously published works on power-balanced VLIW instruction scheduling are relatively scarce. One approach to the problem is to extend the performance-oriented iterative modulo scheduling algorithm by adding heuristics for minimizing processor power variation [14].

In [14], for each instruction k , a power cost p_j^k is associated with instruction k at the j -th pipeline stage. Given a program execution trace W , let W_i represent the set of instructions in the very long instruction executed at the i -th time slot of W . Let n_s denotes the number of pipeline stages. Then power dissipation I_i at the i -th time slot is estimated to be

$$I_i = \sum_{j=1}^{n_s} \sum_{k \in W_{i-j+1}} p_j^k \quad (2.14)$$

Let Π denote the periodic interval of the loop. A software pipelined loop has the minimum power variation when all the I_i ($0 \leq i < \Pi$) are equal to M , which is given by

$$M = \frac{1}{\Pi} \left(\sum_{i=1}^{\Pi} I_i \right) \quad (2.15)$$

The power variation is then given by

$$P = \sum_{i=1}^{\Pi} (I_i - M)^2 \quad (2.16)$$

The instruction-level power model used in [14] is rather simplified. It assumes that each instruction contributes to the total power consumption independently of other instructions. The inter-instruction and intra-instruction effects are not modeled. Also, no validation on the simplified power model regarding the compromised accuracy has been performed.

The power-balanced iterative modulo scheduling algorithm in [14] tries to build balanced schedules using a heuristic guided by the cost function given by (2.16). It is a modified version of the performance-oriented iterative modulo scheduling algorithm. The iterative modulo scheduling algorithm repeats picking the highest priority instruction and then selecting the best time slot in which the instruction is to be scheduled. The performance-oriented algorithm uses the priority function based on the length of the critical dependence cycle and then selects the earliest time slot available in which resource usage and data dependency conflicts are not incurred. In contrast, the power-balanced algorithm uses the modified priority function which schedules instructions with smaller slack widths (i.e. range of its feasible time slots without violating the data dependency) earlier so that performance is not sacrificed. When selecting the best time slot for the current instruction to be scheduled, the cost function (2.16) is used so that the partial schedule is constructed as balanced as possible. In short, two major iterative modulo scheduling decisions are modified so that both performance and power variation are simultaneously considered.

An alternative approach is to formulate it as a mixed-integer program (MIP) [15]. While the heuristic solutions cannot guarantee optimality, solutions of the MIP do. Although algorithms for solving the MIP generally have higher time complexities than

heuristic ones, optimal instruction scheduling only needs to be performed once and the program is executed repeatedly. Therefore code quality is more important than the compilation time. However, the time complexity is particularly high if the program is solved using generic MIP solvers, as is done in [15].

The MIP formulation for power-balanced VLIW instruction scheduling is described as follows. Let $x_i^k = 1$ if instruction k is allocated to time slot i ; otherwise, $x_i^k = 0$. In [15], it was assumed that all pipeline stages in functional unit of type j consume equal power and let it be p^j . Let n_s denote the number of pipeline stages. Let $b_{kj} = 1$ if instruction k uses functional unit type j , otherwise $b_{kj} = 0$. Thus the total amount of consumed power at time slot i is therefore computed as sum of power consumed by the active pipeline stages of all the functional units:

$$I_i = \sum_j \sum_k \sum_{f=1}^{n_s} b_{kf} p^f x_{i-f+1}^k \quad (2.17)$$

Let Π denote the loop interval. A software pipelined loop has the minimum power variation when all the I_i ($0 \leq i < \Pi$) are equal to M , which is given by (2.15). Thus the following objective function is used to measure its power variation:

$$P = \sum_{i=1}^{\Pi} |I_i - M| \quad (2.18)$$

However, they did not provide any validation or references for this VLIW power model. In fact, as shown in the experimentally verified VLIW instruction-level power model described in Section 2.2.2, there are significant differences between the power consumed by the different stages. Thus the optimal schedules obtained based on these rather simplistic power models may be far from optimal in reality.

The MIP formulation in [15] did not incorporate the constraints on data dependency among the target instructions. Besides, it also used a very simplistic resource usage model for their target VLIW processors. Let c_j denote the number of functional units of type j . Let $b_{kj} = 1$ if instruction k uses functional unit type j , otherwise $b_{kj} = 0$.

Constraint on functional unit usage is expressed solely by the following inequality:

$$\sum_k b_{kj} x_i^k \leq c_j \quad \forall j \quad (2.19)$$

In other words, they assume that the instructions scheduled onto each functional unit were executed on dedicated resources in a fully pipelined manner. In fact, instructions with multi-cycle functional unit latency will lock the target functional unit for the necessary number of cycles. No new instruction can be dispatched to that functional unit during this locking period. The second problem is that they do not consider the issue of register utilization. Rescheduling instructions can change variable lifetimes, which may increase pressure on the registers. Without enough registers, the register allocator must insert spill and restore code in the schedule. This may cause extra delay and increase the total energy consumption of the resulting schedule. Processor power variation would also be different. Thirdly, they do not take into account the sharing of resources, such as read/write ports and shared buses between the functional units. As a result, illegal schedules may be obtained.

In Chapters 3 and 4 we shall describe an improved MIP formulation of the problem that makes use of a more accurate power model, together with a more efficient algorithm to solve the problem.

Chapter 3

An Improved Power Model

In this chapter, some enhancements to the power model described in Section 2.2.2 are proposed. They are described in detail in this chapter. The first two modifications (Sections 3.1 and 3.2) are based on a more precise functional-level concurrent activity analysis which improves the accuracy of the model. The last one (Section 3.3) allows us to perform fine-grained run-time power profiling by accounting for the overlapping of instructions within the pipeline while the original model only allows us to compute the total power consumed.

Table 3.1: Power cost of each pipeline step of TI C6201 (200MHz) (from [4]).

	$I_{step}(mA)$			
α	Fetch and DP	DC and E1	E2	E3 to E5
0.125	197.4	60	4	95
0.25	111	64.75	4	95
0.5	69.6	62.5	NA	NA
1	48.125	63.31	NA	NA

Figure 3.1: The improved power model (c.f. Figure 2.4).

3.1 Functional Blocks with Concurrent Activities

As shown in Table 3.1 I_{step} represents the average power cost of each pipeline step relative to the parallelism rate. The power cost of the pipeline steps DC and E1, E2, and E3 to E5 are quite different. Therefore, we propose to divide PU into two blocks. For those instructions that do not involve memory access, the two steps in the decode stage can be grouped with the execute steps as the PU. A new functional block called data memory unit (DMU) is introduced to handle the execution steps for accessing internal data memory.

When the pipeline executes a NOP (no-operation) instruction, it is not dispatched to a functional unit because there is no execution associated with it. Therefore, only the four steps in the fetch stage should be grouped as the IMU; the DP step in the decode stage should be excluded.

The modified functional block grouping is shown in Figure 3.1.

3.2 Instruction-Level Algorithmic Parameters

With the addition of the DMU, an additional algorithmic parameter δ is needed to represent the activity rate of this block and its interactions. Parameter δ which represents the data memory access rate can be computed by

$$\delta = NMU/NMU_{max} \quad (3.1)$$

where NMU_{max} is the maximum number of internal data memory accesses that can be executed in a single instruction cycle and NMU is the average number of internal data memory accesses executed per cycle.

Due to the regrouping, the formulas for computing α and β also need to be modified

accordingly. We now have

$$\alpha = K/K_{max} \quad (3.2)$$

$$\beta = NPU/NPU_{max} \quad (3.3)$$

where K_{max} is the issue width of the target VLIW processor, K is the average width of issued instruction words, NPU_{max} is the maximum number of processing units in the processor and NPU is the average number of processing units used per cycle.

3.2.1 Power Model Coefficients

With the model changes, the expression for the total power I_{tot} , previously given by (2.9), now becomes

$$I_{tot} = I_{IMU} + I_{PU} + I_{DMU} + e \quad (3.4)$$

where I_{IMU} , I_{PU} and I_{DMU} are the power consumed by the blocks IMU, PU and DMU respectively with

$$I_{DMU} = c\delta \quad (3.5)$$

where c is the additional coefficient for the DMU. The power model coefficients can be estimated in the same way as described in Section 2.2.2.3.

3.3 Power Consumption as a Function of Time

Apart from computing the total power, our scheduling problem requires the computation of power consumption as a function of time, i.e. to capture the overlapping of instructions within the pipeline. Let the instruction schedule be $\langle W_1, W_2, \dots, W_i, \dots, W_t \rangle$ where $W_i = (w_i^1, w_i^2, \dots, w_i^k)$ is the long instruction word fetched at the i -th time slot of this schedule and w_i^j , $1 \leq j \leq k$ are the individual instructions that make up W_i . Assume that the i -th time slot of this schedule corresponds to the PS (program address send) pipeline step when W_i is fetched. Then the total power in the i -th time slot of this schedule is

the sum of the power of the IMU, the PU and the DMU in this time slot. It can be expressed mathematically as

$$I^{(i)} = I_{IMU}^{(i)} + I_{PU}^{(i)} + I_{DMU}^{(i)} + e \quad (3.6)$$

The parameters α , β , and δ in (3.2), (3.3), and (3.5) will have to be localized to time slot i , denoted by $\alpha^{(i)}$, $\beta^{(i)}$ and $\delta^{(i)}$ respectively. These localized parameters can be computed by averaging over the consecutive pipeline steps involved. According to Figure 3.1, at the i -th time slot when the instruction word W_i is in its PS (program address send) pipeline step, W_{i-1} is already in the PW (program access ready wait) step and W_{i-2} is in the PR (program fetch packet receive) step in the functional block IMU. So, based on (3.2), we can obtain the localized α by

$$\alpha^{(i)} = \frac{1}{K_{\max}} \left(\frac{K_i + K_{i-1} + K_{i-2}}{3} \right) \quad (3.7)$$

where K_i , K_{i-1} and K_{i-2} are the width of the instruction words in time slots i , $i-1$ and $i-2$, which are at their pipeline steps PS , PW and PR respectively.

Similarly, at the i -th time slot, instruction word W_{i-3} is in the DP (dispatch) step and W_{i-4} is in the DC (decode) step. At the same time, W_{i-5} is in its first execution (E1) pipeline step in the functional block PU. Even if instructions with multiple execution pipeline steps exist, we do not need to add up those instruction words which are in their pipeline steps after E1. This is because, according to experimental data given in [4], the execution steps after E1 in the functional block PU consume much less power compared with the previous steps. For example, there is no significant power dissipation difference between an addition and a multiply instruction because the second execution step E2 which exists for multiply only, consumes only 4 mA per execution against 60 mA per execution of the DC or E1 steps.

For the same reasons, the localized version of β is given by

$$\beta^{(i)} = \frac{1}{NPU_{\max}} \left(\frac{NPU_{i-3} + NPU_{i-4} + NPU_{i-5}}{3} \right) \quad (3.8)$$

where NPU_{i-3} , NPU_{i-4} and NPU_{i-5} are the number of instructions dispatched in time slot i , $i-1$ and $i-2$, which are at their pipeline step DP , DC and $E1$ respectively.

Finally, in the i -th time slot, if any one of W_{i-6} , W_{i-7} and W_{i-8} is involved in the three pipelines steps for data memory access in the functional block DMU. Thus we have

$$\delta^{(i)} = \frac{1}{NMU_{\max}} \left(\frac{NMU_{i-6} + NMU_{i-7} + NMU_{i-8}}{3} \right) \quad (3.9)$$

where NMU_{i-6} , NMU_{i-7} and NMU_{i-8} are the number of internal data memory accesses executed at time slots i , $i-1$ and $i-2$, which are the address send, access ready wait and data receive respectively.

3.4 Power Model of the C6711

The power model coefficients for the C6711 [2, 42] based on the modified power model need to be determined. This processor has already been introduced in Section 2.1.2. The equipment setup and the experimental design are described.

The C6711 requires two power supply rails. The core voltage CV_{DD} supplies power to the processor core while the I/O voltage DV_{DD} supplies the on-chip peripherals. A resistor is inserted on the core power supply (CV_{DD})'s circuit path. The current supplied to the processor core is measured by the voltage drop across a 0.5Ω resistor connected in series with the power supply to CV_{DD} . A high frequency digital oscilloscope is used to perform the measurement. The processor board and the probes are shown in Figure 3.2. Similarly, by inserting a resistor in the I/O power supply's circuit path, we can measure the current supplied to the other on-chip components.

The processor's average current $I^{(i)}$ is measured in order to estimate the values of the instruction-level power model coefficients a , b , c and e . The power model coefficients can be estimated in the same way as described in Section 2.2.2.3. A set of *elementary*

Figure 3.2: Experimental setup for measuring processor current of TMS320C6711.

LOOP :

<i>LDW</i>	<i>.D2</i>	<i>*B5, B1</i>
<i>LDW</i>	<i>.D2</i>	<i>*B5, B1</i>
<i>...</i>	<i>...</i>	
<i>LDW</i>	<i>.D2</i>	<i>*B5, B1</i>
<i>B</i>	<i>.S1</i>	<i>LOOP</i>

Figure 3.3: Example elementary program that activates the PU, IMU and DMU.

programs with various instruction-level algorithmic activities are designed to activate the functional blocks (IMU, PU and DMU).

In addition to those in Section 2.2.2.3, one more example program is shown in Figure 3.3. In the program shown in Figure 3.3, all the three functional blocks are activated. According to the equations (3.7), (3.8) and (3.9), $\alpha^{(i)} = \frac{1}{8}$, $\beta^{(i)} = \frac{1}{8}$ and $\delta^{(i)} = \frac{1}{2}$.

An elementary program q provides the values of its parameters $\alpha_q^{(i)}$, $\beta_q^{(i)}$ and $\delta_q^{(i)}$ and the measured processor's average current $I_q^{(i)}$ while executing this elementary program q . Suppose N elementary program experiments are conducted. Then the power model coefficients a , b , c and e can be estimated by linear regression that minimizes the sum

of squares of the residuals $\sum_{q=1}^N (a\alpha_q^{(i)} + b\beta_q^{(i)} + c\delta_q^{(i)} + e - I_q^{(i)})^2$. In order to cope with the effect of nuisance factors and the imprecision inherent in measurements, it is necessary to produce a large number of elementary program experiments N for the linear regression. The procedure was performed for the measured data using C6711. The values obtained based on our measurements are $a = 617.48$, $b = 445.96$, $c = 528.46$ and $e = 636.1$.

3.5 Summary

The power model described in Section 2.2.2 has been improved and enhanced. The modified model is more accurate and suitable for the power-balanced instruction scheduling problem.

Chapter 4

The Mixed-Integer Programming Approach

In this chapter, we take the MIP approach to the power-balanced VLIW instruction scheduling problem. While this approach has been proposed before, our MIP problem formulation makes use of a much more accurate VLIW instruction-level power model which has been presented in Chapter 3. This MIP also makes use of a more complete resource usage model. It takes into account constraints imposed by multi-cycle functional unit latency as well as limited shared resources such as registers, read/write ports and buses which have not been considered before. An efficient branch-and-bound algorithm is developed to solve the MIP much more efficiently than generic solvers, making the technique more attractive for use in practical compilers.

Experimental results based on Texas Instruments' C6711 [2] VLIW digital signal processor using benchmarks programs from Mediabench [25] and Trimaran [26] showed that over 40% average reduction in power variation can be achieved without degrading the execution performance of the target programs and with a relatively low computational cost.

4.1 Problem Formulation

We assume that the target VLIW processor has a multi-issue, load/store architecture with multiple functional units. A VLIW processor with an issue width of K_{max} has instruction words which are composed of at most K_{max} instructions. The individual instructions are executed simultaneously by different functional units. Each functional unit will have an associated type, which specifies the instruction types it can execute. A functional unit is indexed by the unit type and the index of the particular unit within that type. The number of *delay slots* associated with an instruction is the number of cycles required before the result of that instruction, once issued, is available. *Functional unit latency* is the number of cycles that the instruction engages the functional unit read ports.

The objective is to minimize mean-square power. Due to data dependency and resource usage conflicts among the instructions, not all combinations of instructions are allowed in a single very long instruction word. All legal combinations must satisfy the data dependency constraints and the resource constraints which are imposed by the architecture of the VLIW processor. Furthermore, we also require that all execution deadline must be met. In order to ensure that the execution performance of the target program is not compromised, we can set the execution deadline as the time slots available in the optimal schedule obtained by the performance-oriented VLIW instruction scheduling algorithms. The power-balanced VLIW instruction scheduling problem can be formulated as a mixed-integer program given by **P1** below. For ease of reference, the notations used in the MIP are listed below.

4.1.1 Notations

For ease of reference, the notations used in the rest of this chapter are listed here.

- n is the total number of instructions in the given schedule.
- t is the total number of time slots available.
- $x_i^k = 1$ if instruction k is allocated to time slot i . Otherwise, $x_i^k = 0$.
- X is the set of n variables x_i^k which equal to 1. Thus X specifies a schedule.
- u is the total number of functional unit types in the target VLIW processor.
- c_j is the total number of functional units of type j .
- g is the total number of physical registers.
- s is the total number of shared buses.
- r is the total number of shared read register ports which forbid parallel access.
- w is the total number of shared write register ports which forbid parallel access.
- $a_{qj}^k = 1$ if the q -th functional unit of type j can execute instruction k . It is zero otherwise.
- $b_{kqj}^l = 1$ if instruction k allocated to the q -th functional unit within type j uses shared bus l . It is zero otherwise.
- $d_{qj}^l = 1$ if the q -th functional unit of type j shares read register port l . It is zero otherwise.
- $e_{qj}^l = 1$ if the q -th functional unit of type j shares write register port l . It is zero otherwise.
- E is the set of all flow-dependent pairs $\langle l, m \rangle$, where instruction m depends on instruction l .
- $R_c^l = 1$ if instruction l defines a register variable and it is live at time slot c . Otherwise, $R_c^l = 0$.

- D^k is the number of delay slots for instruction k .
- L^k is the functional unit latency of instruction k .
- $f^k = 1$ if instruction k involves internal data memory access. It is zero otherwise.
- I_i is the average power in the time slot i .
- M is the average power over all t time slots.
- The function $\Lambda(x)$ is defined by

$$\Lambda(x) = \begin{cases} 1, & x \geq 1 \\ 0, & \text{otherwise} \end{cases} \quad (4.1)$$

where x is an integer.

4.1.2 Mixed-Integer Program Formulation

The power-balanced VLIW instruction scheduling problem can be formulated as a mixed-integer program given by **P1** below.

$$\mathbf{P1:} \quad \min P(X)$$

subject to

$$X = \bigcup_{i,k:x_i^k=1} \{x_i^k\} \quad k = 1, \dots, n; i = 1, \dots, t \quad (4.2)$$

$$x_i^k \in \{0, 1\} \quad k = 1, \dots, n; i = 1, \dots, t$$

$$\sum_{i=1}^t ix_i^m - \sum_{i=1}^t ix_i^l \geq D^l \quad \forall \langle l, m \rangle \in E \quad (4.3)$$

$$\sum_{k=1}^n a_{qj}^k x_i^k \leq 1 - \sum_{k=1}^n \Lambda(i - \sum_{z=1}^t x_z^k) \Lambda(\sum_{z=1}^t x_z^k + L^k - i) a_{qj}^k \quad (4.4)$$

$$j = 1, \dots, u; q = 1, \dots, c_j; i = 1, \dots, t$$

$$\sum_{l=1}^n R_c^l \leq g, \quad c = 1, \dots, t \quad (4.5)$$

$$\sum_{k=1}^n \sum_{j=1}^u \sum_{q=1}^{c_j} b_{kqj}^l a_{qj}^k x_i^k \leq 1, \quad l = 1, \dots, s; \quad i = 1, \dots, t \quad (4.6)$$

$$\sum_{k=1}^n \sum_{j=1}^u \sum_{q=1}^{c_j} d_{qj}^l a_{qj}^k x_i^k \leq 1, \quad l = 1, \dots, r; \quad i = 1, \dots, t \quad (4.7)$$

$$\sum_{k=1}^n \sum_{j=1}^u \sum_{q=1}^{c_j} e_{qj}^l a_{qj}^k x_i^k \leq 1, \quad l = 1, \dots, w; \quad i = 1, \dots, t \quad (4.8)$$

$$\sum_{i=1}^t i x_i^k + D^k \leq t, \quad k = 1, \dots, n \quad (4.9)$$

$$\sum_{i=1}^t x_i^k = 1, \quad k = 1, \dots, n \quad (4.10)$$

For this MIP, the decision variables x_i^k are binary. x_i^k equals 1 if instruction k is allocated to time slot i ; otherwise it is 0. A schedule X is represented by the set of non-zero binary variables.

4.1.2.1 Objective function

The objective is to minimize power consumption variation throughout the duration of the VLIW program. Power consumption in each of the t time slots is obtained through using equations (3.6)-(3.9). We need to relate the parameters $\alpha^{(i)}$, $\beta^{(i)}$ and $\delta^{(i)}$ to the binary decision variables in order to derive the objective function for this MIP.

To obtain $\alpha^{(i)}$, K_i , K_{i-1} and K_{i-2} , which are the width of the instruction words in time slot i , $i - 1$ and $i - 2$ respectively, are required. They can be computed by adding up the binary decision variables for time slots i , $i - 1$ and $i - 2$ respectively.

$$\begin{aligned} K_i &= \sum_{k=1}^n x_i^k \\ K_{i-1} &= \sum_{k=1}^n x_{i-1}^k \\ K_{i-2} &= \sum_{k=1}^n x_{i-2}^k \end{aligned} \quad (4.11)$$

For $\beta^{(i)}$, we need to compute NPU_i , NPU_{i-1} and NPU_{i-2} which are the number of instructions dispatched in time slots i , $i-1$ and $i-2$ respectively. They are actually instructions already fetched in time slots $i-3$, $i-4$ and $i-5$ respectively. Therefore,

$$\begin{aligned} NPU_i &= \sum_{k=1}^n x_{i-3}^k \\ NPU_{i-1} &= \sum_{k=1}^n x_{i-4}^k \\ NPU_{i-2} &= \sum_{k=1}^n x_{i-5}^k \end{aligned} \quad (4.12)$$

NMU_i , NMU_{i-1} and NMU_{i-2} which are the number of internal data memory accesses executed at time slots i , $i-1$ and $i-2$ are needed for $\delta^{(i)}$. They are actually instructions already fetched in time slots $i-6$, $i-7$ and $i-8$ respectively. Hence,

$$\begin{aligned} NMU_i &= \sum_{k=1}^n f^k x_{i-6}^k \\ NMU_{i-1} &= \sum_{k=1}^n f^k x_{i-7}^k \\ NMU_{i-2} &= \sum_{k=1}^n f^k x_{i-8}^k \end{aligned} \quad (4.13)$$

Substituting (4.11)-(4.13) into (3.6)-(3.9), the total power consumption I_i in time slot i can be expressed in terms of the binary decision variables x_i^k .

$$\begin{aligned} I_i &= \frac{a}{K_{\max}} \cdot \frac{1}{3} \left(\sum_{k=1}^n x_i^k + \sum_{k=1}^n x_{i-1}^k + \sum_{k=1}^n x_{i-2}^k \right) \\ &+ \frac{b}{NPU_{\max}} \cdot \frac{1}{3} \left(\sum_{k=1}^n x_{i-3}^k + \sum_{k=1}^n x_{i-4}^k + \sum_{k=1}^n x_{i-5}^k \right) \\ &+ \frac{c}{NMU_{\max}} \cdot \frac{1}{3} \left(\sum_{k=1}^n f^k x_{i-6}^k + \sum_{k=1}^n f^k x_{i-7}^k + \sum_{k=1}^n f^k x_{i-8}^k \right) + e \end{aligned} \quad (4.14)$$

Thus the average power over all t time slots is

$$M = \frac{1}{t} \left(\sum_{i=1}^t I_i \right) \quad (4.15)$$

The total power variation for a schedule X is the total amount of deviation from the average value M for all t time slots. To overcome the problem caused by negative differences, define

$$P(X) = \sum_{i=1}^t (I_i - M)^2 \quad (4.16)$$

This is the objective function we seek to minimize.

4.1.2.2 Data flow dependency constraints

Flow dependencies occur when one instruction m uses the result of another l . The time slot when the result of instruction l becomes available is given by $\sum_{i=1}^t ix_i^l + D^l$. Thus instruction m can only be scheduled at this time slot or later. These constraints are specified by (4.3) in the MIP.

Register reuse can introduce additional flow dependencies. Anti-dependencies and output dependencies will not be considered since they can be addressed by register rename. Register pressure will be modeled using the resource usage constraints.

4.1.2.3 Resource usage constraints

Constraints on four different types of resources are modeled by **P1**. We shall consider each of them in detail.

Constraint on functional unit There are two types of constraints on the functional units. First, two instructions that use the same functional unit cannot exist in the same instruction word. For example, the very long instruction of C6711 below is invalid.

$$\begin{array}{l} \text{ADD} \quad .S1 \quad A0, A1, A2; \\ || \quad \text{SHR} \quad .S1 \quad A3, 15, A4; \end{array} \left\{ \begin{array}{l} .S1 \text{ is used for} \\ \text{both instructions} \end{array} \right.$$

Second, if an instruction has a multi-cycle functional unit latency, it locks the functional unit for the necessary number of cycles. No new instruction can be dispatched to that functional unit during this locking period. For this reason, even if an instruction does not have functional unit usage conflict with other instructions in the same instruction word, it still has to be scheduled to a later instruction word. For example, an instruction of the following types of C6711 scheduled on cycle i has the following constraints:

<i>DP compare</i>	<i>No other instruction can use the functional unit on cycles i and $i + 1$</i>
<i>MPYI</i>	<i>No other instruction can use the functional unit on cycles $i, i + 1, i + 2$, and $i + 3$</i>

Let the functional unit be the q -th one of type j . Consider time slot i . First, at most one instruction can be allocated to this functional unit in the current time slot. Mathematically, this is expressed as

$$\sum_{k=1}^n a_{qj}^k x_i^k \leq 1 \quad (4.17)$$

Second, we need to check whether this functional unit is locked by an earlier instruction. This happens when the following conditions are satisfied.

1. The q -th functional unit within type j can execute instruction k , i.e. $a_{qj}^k = 1$.
2. Instruction k is issued in a time slot earlier than i . This can be ensured by

$$\Lambda \left(i - \sum_{z=1}^t x_z^k \right) = 1 \quad (4.18)$$

3. Time slot i is in the locking period of instruction k . This can be ensured by

$$\Lambda \left(\sum_{z=1}^t x_z^k + L^k - i \right) = 1 \quad (4.19)$$

Combining these two functional unit constraints for a given time slot, we have the constraint as in (4.4).

Constraint on registers Rescheduling instructions can change variables' lifetimes, which may increase register pressure. Therefore we need to analyze the lifetime of each register variable to ensure that there are sufficient registers for each time slot. Without loss of generality, we shall assume that

1. all registers are of the same type;
2. every operand of an instruction occupies at most one register;
3. the result of an instruction resides in the register used by later instructions.

Thus we can derive the register usage chain from the dependence relations in the data flow graph. The lifetime of a register variable starts at the time when an instruction defines it until the time when it is last used by other instructions. If instruction l does not define any register variable, let $R_c^l = 0$ for any time slot c . Consider in the data flow graph only one instruction m uses the register variable instruction l has defined. Then

$$R_c^l = \sum_{i=1}^{c-D^l+1} x_i^l - \sum_{i=1}^{c-L^m+1} x_i^m \quad (4.20)$$

is used to test whether the time slot c is in the lifetime of the defined register variable or not. In the case that the time slot c is in the lifetime, instruction l should define it no later than the time slot c , i.e. $\sum_{i=1}^{c-D^l+1} x_i^l = 1$, and instruction m should use it no earlier than the time slot c , i.e. $\sum_{i=1}^{c-L^m+1} x_i^m = 0$. Thus $R_c^l = 1$.

Similarly, if there are more than one instruction that use the register variable instruction l defines,

$$R_c^l = \Lambda \left(\sum_{\forall m \in S} \left[\sum_{i=1}^{c-D^l+1} x_i^l - \sum_{i=1}^{c-L^m+1} x_i^m \right] \right) \quad (4.21)$$

where S denotes the set of instructions that use this register variable. $R_c^l = 1$ indicates that the register variable defined by instruction l is live in time slot c , i.e. the register variable occupies a physical register in time slot c .

Therefore, in any time slot c , R_c^l is used with every instructions to count the total number of the live register variables in the current time slot. The constraint (4.5) in the MIP is to ensure that the total number of register variables that are live in a given time slot does not exceed the total number of registers available.

Constraint on shared buses Instructions in the same instruction word cannot use the same shared buses. For example, the TI C6000 series DSP architectures provide cross path access [2]. Units in one data path may read a single operand from another data path by using a cross path. This is denoted by an X following the functional unit name in the instruction syntax. Hence we cannot schedule two instructions using the same cross path in a single instruction word. For example, the following very long instruction of C6711 is invalid:

$$\begin{array}{ll} \text{ADD} & .L1X \quad A0, B1, A1; \\ || & \text{MPY} & .M1X \quad A4, B4, A5; \end{array} \left. \vphantom{\begin{array}{ll} \text{ADD} & .L1X \quad A0, B1, A1; \\ \text{MPY} & .M1X \quad A4, B4, A5; \end{array}} \right\} \begin{array}{l} 1X \text{ cross path is used} \\ \text{for both instructions} \end{array}$$

Thus a shared bus l should not be used by more than one instructions in any time slot i . An instruction k would use shared bus l in time slot i if this instruction is allocated to time slot i , i.e. $x_i^k = 1$, and it can be executed on the q -th functional unit of type j that requires shared bus l , i.e. $a_{qj}^k b_{kqj}^l = 1$. The constraints (4.6) on shared bus usage is to ensure that the total number of instructions in a given time slot that use the same shared bus must not exceed one.

Constraint on read/write ports If two functional units share a read register port that forbids parallel access for source operands, then instructions using these functional units cannot be scheduled to the same instruction word if they both require source operands. Similar restriction applies to the write register ports for the destination of the instructions.

For example, because the $.L$ functional unit shares its long read port with the store port in C6711, instructions that read a long value cannot be issued on the $.L$ in the same very long instruction as a store. The following very long instruction is invalid:

$$\begin{array}{ll} \text{ADD} & .L1 \quad A5 : A4, A1, A3 : A2; \\ || & \text{STW} & .D1 \quad A8, *A9; \end{array} \left. \vphantom{\begin{array}{ll} \text{ADD} & .L1 \quad A5 : A4, A1, A3 : A2; \\ \text{STW} & .D1 \quad A8, *A9; \end{array}} \right\} \begin{array}{l} \text{long read instruction and a} \\ \text{store} \end{array}$$

Because the $.S$ and $.L$ units share a write register port for long results in C6711, only

one long result may be issued per register file in a very long instruction. The following very long instruction is invalid:

$$\left. \begin{array}{l} \text{ADD} \quad .L1 \quad A5 : A4, A1, A3 : A2; \\ \text{SHL} \quad .S1 \quad A8, A9, A7 : A6; \end{array} \right\} \begin{array}{l} \text{Two long writes} \\ \text{on A register file} \end{array}$$

Thus An instruction k would use read register port l in the time slot i if it is allocated to time slot i , i.e. $x_i^k = 1$, and it can be executed on the q -th functional unit of type j which shares read register port l , i.e. $a_{qj}^k d_{qj}^l = 1$. Thus we arrive at constraints (4.7) and (4.8).

4.1.2.4 Other constraints

Constraint (4.10) ensures that each instruction must be issued once and only once. Constraint (4.9) guarantees that all execution deadline must be met. In order to ensure that the execution performance of the target program is not compromised, we can set the execution deadline t in (4.9) as the time slots available in the optimal schedule obtained by the performance-oriented VLIW instruction scheduling algorithms.

4.1.2.5 Total power

The total power can be assumed as unchanged during rescheduling instructions. This is because constraint (4.9) has fixed the execution deadline and instructions can not be rescheduled beyond it. Furthermore, the equation (2.9) used to compute the total power is linear. Thus rescheduling instructions is actually redistributing their power consumption between the time slots. While the power variation may be changed, the total power computed by (2.9) always keeps same.

4.2 A Branch-and-Bound Algorithm

The complexity of the exhaustive search space for the solution to **P1** is 2^{nt} , for a total of n instructions and t time slots. Generic MIP solvers are generally much more efficient than exhaustive search. However, they do not have problem-specific knowledge to reduce the search space and therefore are inferior to an algorithm that incorporates such knowledge. Therefore we develop a branch-and-bound algorithm specifically for this problem so that optimal solutions can be found efficiently.

4.2.1 Preliminaries

We shall assume that an initial schedule is obtained by using a standard performance optimized compiler. Therefore the task is to reschedule for power variation minimization. Starting with the initial schedule X_1 , a branch-and-bound tree can be constructed using X_1 as the root node. Each node of the tree represents a feasible schedule. A branch of the tree represents a pair of feasible schedules (X_r, X_t) such that X_t is obtained from X_r by rescheduling one single instruction.

For a given schedule X_r in the branch and bound tree, we define two associate sets of instructions. The first set, denoted U_r , consists of the instructions rescheduled along the path from X_1 to X_r . The second set, denoted V_r , consists of all the remaining instructions yet to be rescheduled. Then we say that a schedule X_s is a *successor* of the schedule X_r if we can obtain X_s from X_r by rescheduling only the instructions in the set V_r .

Given a schedule X_r , let I_i^u denote the total power consumption in time slot i due to the instructions in U_r . The formula for calculating I_i^u can be obtained by modifying

(4.14). It is given by

$$\begin{aligned}
 I_i^u = & \frac{a}{K_{\max}} \cdot \frac{1}{3} \left(\sum_{\forall k \in U_r} x_i^k + \sum_{\forall k \in U_r} x_{i-1}^k + \sum_{\forall k \in U_r} x_{i-2}^k \right) + \\
 & \frac{b}{NPU_{\max}} \cdot \frac{1}{3} \left(\sum_{\forall k \in U_r} x_{i-3}^k + \sum_{\forall k \in U_r} x_{i-4}^k + \sum_{\forall k \in U_r} x_{i-5}^k \right) + \\
 & \frac{c}{NMU_{\max}} \cdot \frac{1}{3} \left(\sum_{\forall k \in U_r} f^k x_{i-6}^k + \sum_{\forall k \in U_r} f^k x_{i-7}^k + \sum_{\forall k \in U_r} f^k x_{i-8}^k \right) + e
 \end{aligned} \tag{4.22}$$

Every branch and bound algorithm has three basic elements which determine its efficiency and effectiveness. They are:

1. the rules for selecting one from the current pool of leaf schedules;
2. the rules for generating a set of new schedules from the selected leaf schedule; and
3. a lower bound estimate for the objective function values of all the successors generated from the selected leaf schedule.

In the following sections, the three main elements of our customized branch and bound algorithm are described in detail.

4.2.2 Branching

Let X_{CBS} denote the current best schedule. For a selected leaf schedule X_r , if the lower bound of the objective function values for all the successors of X_r is not less than the objective function value for X_{CBS} (i.e. $P(X_{CBS})$), then it indicates that no better schedule can be found in the successors of X_r . Therefore there is no need to branch from X_r and X_r can be removed from the leaf schedule pool. Otherwise we branch from X_r .

New schedules are generated from X_r by rescheduling an instruction in the set V_r . Thus the branching algorithm needs to determine which instruction in the set V_r to

reschedule. In choosing the instruction to reschedule, we impose the selection rules presented in Section 4.2.3. The priority of rescheduling the total n instructions determined by these selection rules is stored in the array `RescheduleInstructionSequence[n]`.

Given the selected leaf schedule X_r , suppose instruction k in V_r is to be rescheduled. First, we branch from X_r by rescheduling instruction k to all time slots. We check the generated schedules against the constraints (4.3- 4.8). The objective function values of those feasible ones are then computed. If any feasible schedule X_f has an objective function value smaller than $P(X_{CBS})$, then X_f becomes the new X_{CBS} . Next, we delete X_r from the leaf schedule pool and the generated feasible schedules are examined to see which ones are to be inserted into this pool. Let X_s be the new schedule after instruction k is rescheduled to time slot j . Then X_s can be inserted into this pool if it satisfies the following five conditions. The branching process for a selected leaf schedule is summarized in Algorithm 1.

Condition 1 I_j^u should not be larger than the peak power of the current best schedule.

That is,

$$I_j^u \leq P_{max}(X_{CBS}) \quad (4.23)$$

where $P_{max}(X_{CBS})$ is the peak power of the current best schedule.

Condition 2 The data dependency graph is expressed in the form of the data dependency constraints (4.3), which specify the execution order of the instructions. Let $L_{a,b}$ be the length of the longest path from instruction a to instruction b in the data dependence graph. Further, let

$$G_{a,b} = \begin{cases} L_{a,b}, & \text{if instruction } b \text{ directly or indirectly depends on instruction } a \\ -1, & \text{otherwise} \end{cases} \quad (4.24)$$

```

input : The selected leaf schedule  $X_r$ .
output: The updated leaf schedule pool.

1 if the lower bound for  $X_r <$  the objective function value of the
   CurrentBestSchedule then
2    $i \leftarrow$  the depth of  $X_r$ ;
3   Reschedule the instruction at RescheduleInstructionSequence[ $i+1$ ] to all the
   time slots;
4   if the objective function value of any resulting feasible schedule  $<$  the
   objective function value of the CurrentBestSchedule then
5     Replace CurrentBestSchedule with this schedule;
6   end
7   if the depth of  $X_r < n - 1$  then
8     /*Otherwise the branching process will reach the bottom of the branch
       and bound tree.*/;
9     Reschedule the instruction at RescheduleInstructionSequence[ $i+1$ ] to the
       time slots constrained by the five conditions described in Section 4.2.2;
10    Add the generated schedules to the leaf schedule pool;
11  end
12 end
13 Delete  $X_r$  from the leaf schedule pool.

```

Algorithm 1: The branching process for a selected leaf schedule X_r . (RescheduleInstructionSequence[n] is an array to store the priority of rescheduling the total n instructions determined by the selection rules in Section 4.2.3.)

Then, constraints (4.3) can be expressed as

$$\sum_{i=1}^t ix_i^b - \sum_{i=1}^t ix_i^a \geq G_{a,b} \quad (4.25)$$

for $a, b = 1, 2, \dots, n$ and $G_{a,b} > 0$. Condition 2 is that in the new schedule X_s , rescheduling instruction k to time slot j should satisfy constraint (4.25). That is, given $G_{a,k}, G_{k,b} \neq -1, \forall a, b \in U_r$

$$\begin{aligned} j - \sum_{i=1}^t ix_i^a &\geq G_{a,k} \\ \sum_{i=1}^t ix_i^b - j &\geq G_{k,b} \end{aligned} \quad (4.26)$$

Condition 3 The time slots where an instruction may be scheduled must satisfy the constraints (4.25) as well as the deadline constraints (4.9). For instruction k in the new schedule X_s , the range of time slots are defined by

$$\sum_{j=1}^{G_{front}^k} x_j^k = 0, \quad \text{if } G_{front}^k \geq 1 \quad (4.27)$$

$$\sum_{j=G_{front}^k+1}^{t-G_{back}^k+1} x_j^k = 1 \quad (4.28)$$

$$\sum_{j=t-G_{back}^k+2}^t x_j^k = 0, \quad \text{if } G_{back}^k \geq 1 \quad (4.29)$$

where

$$G_{front}^k = \max_h(G_{h,k})$$

$$G_{k,F} = \max_l(G_{k,l})$$

$$G_{back}^k = G_{k,F} + D^F$$

and D^F is the number of time slots needed to execute instruction F .

Condition 4 The resource constraints (4.4-4.8) must be satisfied in time slot j .

Condition 5 The lower bound of the objective function values for the successors of X_s must be less than $P(X_{CBS})$.

4.2.3 Selection

Given the current pool of leaf schedules, the branch and bound algorithm uses the selection rules to choose the one for branching. The selection strategy is to give higher priority to the leaf schedule with less instructions in its U_r . Such a selection rule ensures that the instructions with less data dependencies are rescheduled earlier. Instructions with less data dependency may have more schedule slacks for power balancing. Therefore, it would accelerate the convergence of the branch and bound algorithm if these schedule slacks are explored earlier.

4.2.4 Lower Bound Estimation

The efficiency and effectiveness of the branch and bound algorithm is highly dependent on how accurately the lower bound of the objective function, given a selected schedule X_r , can be estimated. There are two basic requirements for the lower bound estimation algorithm. First, the estimated lower bound should be tight, i.e. its value should not be too far off the optimal value for the successors of X_r . This helps to reduce the search space. Second, it should be computationally efficient.

A common approach to the estimation of a lower bound is to relax the integer constraints on the non-branched decision variables in the MIP. The optimal solution of the relaxed program then provides a lower bound estimation. However, a direct application of this approach to **P1** is not computationally efficient enough since it still involves solving the original program albeit without the integer variables. More importantly, this approach does not provide us with a lower bound that is tight enough for our problem.

Our approach is to recognize that at this time we are only interested in obtaining a lower bound of the objective function values instead of an optimal instruction schedule. In any successor of X_r , the power consumption at time slot i , denoted by I_i , is a sum of the power consumption of instructions already rescheduled (I_i^u) plus the power consumption of the instructions from the set V_r . By redistributing the power consumption of instructions in V_r , we obtain the minimum of the objective function (4.16), which is a lower bound of the objective function values for the successors of X_r .

The total power consumption of each instruction in V_r is composed of power consumption in each function block which is spread out over a number of time slots. For each instruction, we calculate P_{IMU} , P_{PU} and P_{DMU} which is the power consumed in function blocks IMU, PU and DMU respectively *per time slot*, assuming uniform distribution. Let y_i^{IMU} , y_i^{PU} and y_i^{DMU} denote the number of units of P_{IMU} , P_{PU} and P_{DMU} in time slot i respectively. Then we can formulate an alternative mixed-integer program, $\mathbf{P}_{LB}$, based on these variables. The optimal solution of $\mathbf{P}_{LB}$ will provide us with a lower bound of the objective function values for the successors of X_r .

$$\mathbf{P}_{LB}: \quad \min P(X) = \sum_{i=1}^t (I_i - M)^2$$

subject to

$$\begin{aligned} y_i^{IMU} &\geq 0 \\ y_i^{PU} &\geq 0 \\ y_i^{DMU} &\geq 0 \end{aligned} \tag{4.30}$$

$$I_i^v = y_i^{IMU} P_{IMU} + y_i^{PU} P_{PU} + y_i^{DMU} P_{DMU}, \quad i = 1, \dots, t \tag{4.31}$$

$$\sum_{i=1}^t I_i^v = I_{total} - \sum_{i=1}^t I_i^u \quad (4.32)$$

$$I_i = I_i^u + I_i^v, \quad i = 1, \dots, t \quad (4.33)$$

where

$$I_{total} = tM \quad (4.34)$$

The objective function of $\mathbf{P}_{LB}$ is the same as that of $\mathbf{P1}$. But the decision variables are now y_i^{IMU} , y_i^{PU} and y_i^{DMU} , instead of the binary variables x_i^k . I_i^u represents the total power of the instructions that has already been rescheduled (U_r). I_i^v represents the power of the instructions that has yet to be rescheduled (V_r). In solving $\mathbf{P}_{LB}$, we are essentially redistributing the various portions of power of those instructions in V_r , instead of the instructions themselves, to the available time slots. Note that no actual instruction schedules are explicitly obtained.

$\mathbf{P}_{LB}$ implicitly ensures that instructions will not occupy fractions of a time slot by sub-dividing the power of a single instruction in a functional block into integer number of time slots. In contrast, if we simply relax the constraints on the binary decision variables x_i^k of $\mathbf{P1}$, some instructions will not start at the beginning of a time slot. This leads to a solution with power variation much less than that obtained through $\mathbf{P}_{LB}$. So $\mathbf{P}_{LB}$ is able to provide us with a much tighter lower bound. A more formal statement is made in the following theorem.

Theorem 1. *For a given schedule X_r , the optimal value of $P(X)$ obtained by the integer program $\mathbf{P}_{LB}$ is a lower bound of the objective function $P(X)$ of the mixed-integer program $\mathbf{P1}$ for all successors of the schedule X_r .*

Proof. For a given schedule X_r , let s_{LB} denote the optimal value of $P(X)$ for the integer program $\mathbf{P}_{LB}$. We need to prove that for any successor X_s of X_r , $s_{LB} \leq P(X_s)$.

Let $\mathbf{P1}'$ be the subproblem of $\mathbf{P1}$ with the partial schedule X_r . It is the MIP with binary decision variables x_i^k for $k \in V_r$ and $i = 1, \dots, t$. Let $s_{P1'}$ denote the optimal objective function value of $\mathbf{P1}'$. Then for any successor X_s of X_r ,

$$s_{P1'} \leq P(X_s) \quad (4.35)$$

Now compare $\mathbf{P_{LB}}$ with $\mathbf{P1}'$ for a given schedule X_r . The two formulations have the same objective function. But in $\mathbf{P_{LB}}$, the data dependency and resource constraints on the instructions in V_r have been removed. Since $\mathbf{P_{LB}}$ is actually the same problem as $\mathbf{P1}'$ but with less constraints, the optimal value of $\mathbf{P_{LB}}$ must be less than the one obtained by $\mathbf{P1}'$. That is, for the given schedule X_r ,

$$s_{LB} \leq s_{P1'} \quad (4.36)$$

Therefore, based on (4.35) and (4.36), we have $s_{LB} \leq P(X_s)$ for any successor X_s of X_r . $\square$

Furthermore, $\mathbf{P_{LB}}$ is a very simple integer program which can be solved very efficiently. P_{LB} does not have any data dependency and resource constraints which exist in $\mathbf{P1}$. Hence it can be solved by using a simple water-filling algorithm as shown in Figure 2. This algorithm starts with the schedule where each time slot already has power consumption given by I_i^u . The power units of the instructions in V_r are filled into the time slots of this schedule by filling the time slots with lowest power first. Those power units which are larger are chosen first because it is easier to use the smaller portions to “fill in the gaps” later so that power variations are minimized.

4.3 Performance Evaluation

We use the Texas Instruments C6711 digital signal processor as the target VLIW processor for our experiments. Its detailed resource constraints and instruction set information

input : The schedule X_r .

output: The lower bound of the objective function values for all the successor generated from X_r .

```

1  $bound = 0$ ;
2 calculate each  $I_i^u$  according to (4.22); initiate each  $I_i$  with its  $I_i^u$ :  $I_i = I_i^u$ ;
3  $tempCount \leftarrow$  number of units of  $P_{DMU}$  consumed by the instructions in  $V_r$ ;
4 for  $i \leftarrow 1$  to  $tempCount$  do
5     Suppose the value in time slot  $W$  is the smallest among the current  $I_i$ 
      ( $i = 1$  to  $t$ ), then  $I_W = I_W + P_{DMU}$ ;
6 end
7  $tempCount \leftarrow$  the number of units of  $P_{IMU}$  consumed by the instructions in  $V_r$ ;
8 for  $i \leftarrow 1$  to  $tempCount$  do
9     Suppose the value in time slot  $W$  is the smallest among the current  $I_i$ 
      ( $i = 1$  to  $t$ ), then  $I_W = I_W + P_{IMU}$ ;
10 end
11  $tempCount \leftarrow$  the number of units of  $P_{PU}$  consumed by the instructions in  $V_r$ ;
12 for  $i \leftarrow 1$  to  $tempCount$  do
13     Suppose the value in time slot  $W$  is the smallest among the current  $I_i$ 
      ( $i = 1$  to  $t$ ), then  $I_W = I_W + P_{PU}$ ;
14 end
15  $bound \leftarrow$  the objective function value computed from the obtained
       $I_i(i = 1, \dots, t)$ ;
16 Return  $bound$ ;

```

Algorithm 2: A water-filling algorithm to solve the integer program $\mathbf{P}_{LB}$.

Table 4.1: Results of reduction in power variation, computational efficiency and largest power deviation reduction for the Trimaran benchmarks.

Benchmark	Fn ($\times 10^6$)	F ($\times 10^6$)	T (sec.)	fn	f	Imv (%)	ImvT ($\times 10^6$)	Imf (%)
bmm	12.23	5.98	1.86	400	324	51.09	3.36	19.05
dag	1.55	0.953	0.046	290	290	38.57	13.01	0
eight	0.96	0.63	0.031	227	138	34.22	10.56	39.16
fact2	0.65	0.38	0.03	162	115	41.53	8.69	29.45
fib	0.753	0.347	0.046	162	94	53.85	8.81	41.98
fib-mem	1.07	0.504	0.11	213	147	52.87	5.19	30.86
fir	9.28	6.14	1.185	468	423	33.81	2.65	9.66
hyper	0.69	0.374	0.031	156	99	45.68	10.14	36.68
ifthen	1.59	0.988	0.062	290	290	37.83	9.70	0
mm	4.04	2.076	0.498	399	310	48.67	3.95	22.15
mm-double	5.56	3.08	0.72	390	308	44.61	3.45	20.98
mm-dyn	4.14	1.74	0.688	399	229	58.15	3.49	42.55
mm-int	3.42	1.64	0.39	420	287	52.21	4.58	31.56
nested	2.12	0.76	0.16	414	195	64.20	8.71	52.85
sqrt	3.94	2.60	0.69	365	365	33.88	1.95	0
strcpy	1.16	0.61	0.093	224	224	47.31	5.90	0
wave	2.16	0.91	0.20	302	203	57.95	6.21	32.85
Average						46.85	6.49	24.10

can be found in [43]. Our experiments in this section employ the instruction-level power model we built for C6711 in Section 3.4.

The Mediabench benchmarks [25] and the Trimaran benchmarks [26] are used for testing. The benchmark programs are compiled using the compiler in Code Composer Studio with optimization options “-o3” (optimization enabled at file level) and “-ms0” (speed first, size second). Code Composer Studio performs classical compiler optimizations with machine-dependent optimizations and instruction scheduling, producing assembly code. Our branch and bound algorithm is then used for rescheduling the instructions to minimize power variation.

All our computational experiments were conducted on an Intel Pentium 4 2.8 GHz

Table 4.2: Results of reduction in power variation, computational efficiency and largest power deviation reduction for the Media benchmarks.

Benchmark	F_n ($\times 10^6$)	F ($\times 10^6$)	T (sec.)	f_n	f	Imv (%)	$ImvT$ ($\times 10^6$)	Imf (%)
adpcm	3.74	1.84	8.97	384	267	50.85	0.212	30.38
epic	240.99	134.74	188.72	821	718	44.09	0.56	12.49
g721	65.78	35.01	23.47	524	496	46.77	1.31	5.43
gsm	362.23	265.77	191.71	796	631	26.63	0.50	20.68
jpeg	1416.44	781.59	1497.38	883	741	44.82	0.42	16.10
mpeg2	981.26	614.46	808.78	955	775	37.38	0.45	18.80
Average						41.76	0.58	17.31

Figure 4.1: Convergence process of a problem instance with a size of (60, 39).

Table 4.3: Tradeoff between computation time and power variation reduction with maximum subproblem fixed at 300.

Program	F _n	F	T (sec.) ($\times 10^6$)	Imv	ImvT	dImv	uT
($\times 10^6$)	($\times 10^6$)	(%)	($\times 10^6$)	(%)	(%)	(%)	
jpeg	1416	909	490	35.83	1.035	8.99	67.25
mpeg2	981	716	214	26.99	1.237	10.39	73.53
Average						9.68	70.39

personal computer with 512MB RAM running under Microsoft Windows 2000. The results of the two benchmark suits are shown in Tables 4.1 and 4.2. Columns “F_n” and “F” are the amount of power variation of the schedules produced by Code Composer and by our algorithm respectively. Column “Imv” shows the corresponding percentage improvements by comparing “F_n” and “F”. Columns “fn” and “f” show the maximum power deviation from the mean value for the two kinds of schedules. Column “T” is the computation time of the branch and bound algorithm. Column “ImvT” is computed by

$$ImvT = \frac{Fn - F}{T} \quad (4.37)$$

which indicates the gain per unit time.

The results are summarized as follows:

1. *Reduction in power variation:* Overall, our algorithm produces schedules with an average improvement of 46.85% for the Trimaran benchmarks and 41.76% for the MediaBench benchmarks while maintaining the same performance as the non-power balanced schedules.
2. *Reduction of maximum power deviation:* There is a 17.31% reduction (column “Imf”) on average in terms of maximum deviation from the mean.
3. *Computational efficiency:* The improvement is 6.4908×10^6 unit per second for the Trimaran benchmarks and 0.5777×10^6 unit per second for Mediabench.

Furthermore, as a result of the proposed branching rules and selection rules, the branch and bound algorithm is able to reach a solution that is very close to the global optimum very quickly. Figure 4.1 shows the convergence behavior for a program block with 60 instructions and 39 time slots. The size of the exhaustive search space is $2^{60 \times 39}$. However, our branch and bound algorithm is able to reach a schedule with an objective function value within 4.4% of the global optimum before 380 leaf schedules have been visited.

Further experiments are conducted using the most time consuming media benchmarks, “jpeg” and “mpeg2”. We set a maximum limit of 300 nodes per instruction block. Table 4.3 shows the results. The computation time saving percentage is shown in Column “uT”. The degraded best objective function value (column “dImv”) is described by the difference between the improvement percentages (column “Imv”) in Table 4.2 and the improvement percentage in this table. For these two programs, an average saving of 70.39% in computation time can be achieved at the cost of a reduction of improvement by 9.68%.

Finally we compare the required computation time between the proposed branch-and-bound algorithm and the general mixed integer optimizer in ILOG CPLEX as it was used in [15]. The results reported in Table 4.4 are given for single problem instances taken from Trimaran benchmark programs. The Column ‘Dimension’ indicates the number of available time slots and the number of instructions to be scheduled. The total CPU time in seconds required by the branch and bound algorithm and the general mixed integer optimizer in ILOG CPLEX are given in the Columns ‘T’ and ‘CPLEX’ respectively. We can see that the computation time required by the general mixed integer optimizer is far from acceptable, particularly if it is used for the multimedia programs with big sizes.

Table 4.4: Comparison of computation time between the proposed branch-and-bound algorithm and the mixed integer optimizer in ILOG CPLEX

Dimension	Trimaran Benchmark	T(sec.)	CPLEX(sec.)
(6,14)	Wave	0.03	0.05
(11,11)	Fib	0.01	0.07
(9,14)	Wave	0.11	0.13
(13,13)	Wave	0.01	0.07
(10,22)	Bmm	0.10	0.19
(15,15)	Fib-mem	0.01	0.10
(12,19)	Fir	0.06	0.10
(12,22)	Bmm	0.54	2.16
(17,16)	Fib-mem	0.03	0.57
(20,21)	Wc	0.18	0.22
(13,35)	Bmm	0.47	0.58
(23,22)	Bmm	0.09	5.43
(23,24)	Bmm	0.06	0.27
(25,24)	Bmm	0.34	19.29
(29,29)	Bmm	0.04	1.44
(31,30)	Bmm	0.26	10.04
(33,33)	mm-dyn	0.85	1.96
(35,34)	mm-dyn	1.59	108.47

4.4 Summary

The problem of power-balanced VLIW instruction scheduling is formulated as a mixed-integer program. While this approach has been proposed before, our contributions are two-fold. Firstly, an efficient branch-and-bound algorithm is developed specifically for this problem, enabling it to be solved much more efficiently than generic techniques. Secondly, a more complete resource usage model and a more accurate VLIW instruction-level power model are incorporated into the problem. The resource usage model takes into account constraints imposed by multi-cycle functional unit latency as well as shared resources such as limited registers, read/write ports and buses which have not been considered before. The power modelling method applied to TI C67 with classical digital signal processing algorithms shows that the maximum error between the estimates and the physical measurements is 6 percent in [4]. The results of simulation experiments based on the C6711 VLIW digital signal processor using benchmarks programs from Mediabench and Trimaran confirmed the effectiveness and efficiency of our methods.

The techniques proposed here are for VLIW instruction scheduling at the compile time for minimum power variation of the target processor. However, the power variation during the execution of the program would be affected by instruction and data cache misses which are not considered here.

Chapter 5

Basic Concepts of Rough Sets, Rough Variables and Rough Program

In this chapter, we introduce some basic concepts of rough sets, rough variables and rough programming that will be needed to understand our research work presented in Chapters 6 and 7.

5.1 Rough Sets

The notion of rough set [44–47] is designed to deal with imprecise concepts. Given a set of attributes S that objects in a universe U possess, two objects $x, y \in U$ are indiscernible from each other if

$$s(x) = s(y), \quad \forall s \in S \quad (5.1)$$

where $s(x)$ and $s(y)$ denote the values of attribute s . In this case, x and y are said to have an indiscernible relation $I(S)$ such that (5.1) can alternatively be expressed as

$$x \ I(S) \ y \quad (5.2)$$

Therefore $I(S)$ is an equivalence relation that will partition U into equivalence classes. A set of the objects in a equivalence class is called an *elementary set* and the union of all elementary sets is called an *approximation space* $A = (U, I(S))$.

Any subset $Y \subset U$ is called a *concept*. When the objects in the same elementary set belong to different concepts, then they are not definable using crisp elementary sets. Rough set offers two approximations, known as the lower and upper approximation, to define such a concept Y . The lower approximation of Y is defined as the greatest union of the elementary sets contained in Y and the upper approximation of Y is defined as the least union of the elementary sets containing Y . Mathematically, these approximations, denoted by $I_*(Y)$ and $I^*(Y)$ respectively, are given by

$$I_*(Y) = \{y \in U \mid I(y) \subseteq Y\} \quad (5.3)$$

$$I^*(Y) = \{y \in U \mid I(y) \cap Y \neq \emptyset\} \quad (5.4)$$

where $I(y)$ denotes the elementary set containing the object y . Thus all the objects in the lower approximation are certainly valid for describing the concept Y . But those objects in the upper approximation are only possibly valid descriptions of the concept. The difference between the upper and lower approximations constitute the boundary region of Y . The approximation accuracy of Y in the approximation space $A = (U, I(S))$ can be numerically characterized as,

$$\mu^A(Y) = \frac{\text{card}(I_*(Y))}{\text{card}(I^*(Y))}$$

The larger the value of $\mu^A(Y)$, the more accurate the rough set approximation of Y is. In this way, we can characterize the vagueness of the concept.

The idea of rough sets proves to be very useful in practice. Very promising results using rough sets for classification have been obtained in domains such as handwritten numerical character recognition [48], signal classification [49, 50], software quality analysis [51], and economic and financial prediction [52].

5.2 Rough Variables

A rough variable [24] may be viewed as a rough set in a continuous-valued approximation space. That is, given an object x and attribute s , $s(x)$ is continuous-valued. Then, using the definition (5.1), there will be few objects in U which are indiscernible and so the number of equivalence classes defined by $I(S)$ will be prohibitively large. This will lead to a large number of elementary sets each with very few elements. One way to overcome this problem is to note that the concepts are imprecise. Thus we can discretize the real numbers into a finite number of intervals. This results in a new universe U' where the real-valued attribute values of the objects are replaced by the intervals they belong to. Then the indiscernibility relation $I(S)$ between the objects in U' can now be used to partition U' into the equivalence classes, resulting in a new approximation space $A' = (U', I(S))$.

In our problem, the concepts we want to represent are the coefficients of a VLIW power model. Hence the only attribute is “power” w which is real-valued. In such one-attribute approximation spaces, U is actually a set of real numbers and U' is a set of intervals on the real number line. Rough variables in the approximation space $(U', I(w))$ can be represented in a form $([a, b], [c, d])$ with $c \leq a \leq b \leq d$, where the lower approximation $[a, b]$ and upper approximation $[c, d]$ are closed intervals of attribute w [24].

5.2.1 Discretization

Let $[l, r] \in \mathfrak{R}$ be the domain of attribute w . Any $b \in [l, r]$ defines a partition of $[l, r]$ into two intervals and is known as a cut on $[l, r]$. A set of cuts on $[l, r]$, $B = \{b_1, b_2, \dots, b_m\}$ partition $[l, r]$ into subintervals $[b_0, b_1], [b_1, b_2], \dots, [b_m, b_m + 1]$, where $b_0 = l$ and $b_m + 1 = r$. The *optimal consistent* cut set $B = \{b_1, b_2, \dots, b_m\}$ should satisfy

- Preserve the discernibility between measurement results from different coefficients:
 $\forall x, y \in U$, if x and y are surely measurement results for different power model

coefficients, then x and y in U' should be assigned different intervals;

- Optimization criteria are defined by number of cuts: For any cut set B' satisfying the first requirement, we have $\text{card}(B) \leq \text{card}(B')$.

Finding the optimal set of cuts for a given set of data is known to be NP-hard. Sub-optimal solutions can be obtained by using the entropy/MDL algorithm and the Boolean reasoning algorithm [53–55]. The Boolean reasoning algorithm is the most popular choice for rough sets. It translates the original problem of searching for semi-optimal set of cuts into one that computes the minimal prime implicant of a Boolean function [54].

The Boolean reasoning algorithm first naively generates the B which simply consists of all cuts midway between two measurement results, except for the cuts which are clearly not needed because we do not need to discern two measurement results surely for the same coefficient. The set of naively generated cuts preserves the original discernibility between measurement results from different coefficients. Then in order to reduce this initial set to its minimal subset, each cut b_i is assigned a Boolean variable a_i . The Boolean function $h^{x,y} = \vee\{a_i | x \leq c_i < y\}$ is defined to discern two measurement results x and y surely for different coefficients. The minimal B can make $h^{x,y}$ true only if the Boolean variable of any cut in B is true in $h^{x,y}$. Thus the Boolean function $h = \wedge h^{x,y}$ is defined for all $x, y \in U$. The optimal B is the set cuts where their corresponding Boolean variables are a prime implicant of h .

Then the original U is transformed into U' by, for $\forall x \in U$, replacing $x = d$ with $x = [b_i, b_{i+1})$ where $d \in [b_i, b_{i+1})$. Then $\forall x, y \in U'$ are indiscernible from each other if they are assigned with the same interval. This indiscernible relation partitions U' into equivalence classes. Let $I(y)$ denote the equivalence class containing y . The rough

variable $([a, b], [c, d])$ that represents Z is obtained by

$$[a, b] = \cup \{Y_y \mid y \in U' \text{ and } I(y) \subseteq Z\} \quad (5.5)$$

$$[c, d] = \cup \{Y_y \mid y \in U' \text{ and } I(y) \cap Z \neq \emptyset\} \quad (5.6)$$

where Y_y denotes the interval y belongs to. Thus all the values in $[a, b]$ are valid description of Z with certainty and the values in $[c, d]$ are valid description of Z possibly. The difference between the upper and lower approximations constitute the boundary region of Z . The approximation accuracy of Z in the approximation space $A' = (U', I(S))$ can be numerically characterized as,

$$\mu^{A'}(Z) = \frac{b - a}{d - c}$$

The larger the value of $\mu^{A'}(Z)$, the more accurate the rough variable approximation of Z is. In this way, we can characterize the vagueness of the concept.

5.2.2 Basic Rough Arithmetic

Let $\zeta = ([a_1, a_2], [a_3, a_4])$ and $\eta = ([b_1, b_2], [b_3, b_4])$ be two rough variables. Given that $a_3 \leq a_1 \leq a_2 \leq a_4$ and $b_3 \leq b_1 \leq b_2 \leq b_4$, the following rules for rough arithmetic has been defined

$$\zeta + \eta = ([a_1 + b_1, a_2 + b_2], [a_3 + b_3, a_4 + b_4]) \quad (5.7)$$

$$\zeta - \eta = ([a_1 - b_2, a_2 - b_1], [a_3 - b_4, a_4 - b_3]) \quad (5.8)$$

$$k\zeta = \begin{cases} ([ka_1, ka_2], [ka_3, ka_4]) & \text{for } k \geq 0 \\ ([ka_2, ka_1], [ka_4, ka_3]) & \text{for } k < 0 \end{cases} \quad (5.9)$$

$$\zeta \times \eta = ([a_1 \times b_1, a_2 \times b_2], [a_3 \times b_3, a_4 \times b_4]) \text{ for } a_3 \geq 0, b_3 \geq 0 \quad (5.10)$$

$$\zeta \div \eta = ([a_1 \div b_2, a_2 \div b_1], [a_3 \div b_4, a_4 \div b_3]) \text{ for } a_3 \geq 0, b_3 > 0 \quad (5.11)$$

5.2.3 Ordering Rough Values

Unlike integers or real numbers, there is no natural way to order the rough values. There are several different ways to rank the values of a rough variable of the form $([a, b], [c, d])$. We may use the trust measure, α -pessimistic value, α -optimistic value or the expected value [24].

Definition 2. Let $\zeta = ([a, b], [c, d])$ be a rough variable with $c \leq a \leq b \leq d$. Then the trust of $\zeta \leq r$ is given as

$$Tr\{\zeta \leq r\} = \begin{cases} 0, & r \leq c \\ \frac{1}{2} \left(\frac{c-r}{c-d} \right), & c \leq r \leq a \\ \frac{1}{2} \left(\frac{c-r}{c-d} + \frac{a-r}{a-b} \right), & a \leq r \leq b \\ \frac{1}{2} \left(\frac{c-r}{c-d} + 1 \right), & b \leq r \leq d \\ 1, & r \geq d \end{cases} \quad (5.12)$$

The trust of $\zeta \geq r$ is given as

$$Tr\{\zeta \geq r\} = \begin{cases} 1, & r \leq c \\ \frac{1}{2} \left(\frac{r-d}{c-d} + 1 \right), & c \leq r \leq a \\ \frac{1}{2} \left(\frac{r-d}{c-d} + \frac{r-b}{a-b} \right), & a \leq r \leq b \\ \frac{1}{2} \left(\frac{r-d}{c-d} \right), & b \leq r \leq d \\ 0, & r \geq d \end{cases} \quad (5.13)$$

We say two rough values $\zeta < \eta$ if and only if, for some predetermined r , we have $Tr\{\zeta < r\} > Tr\{\eta < r\}$.

Definition 3. Let $\zeta = ([a, b], [c, d])$ be a rough variable with $c \leq a \leq b \leq d$, and $\alpha \in (0, 1]$. Then

$$\zeta_{\inf}^{\alpha} = \inf \{r | Tr\{\zeta \leq r\} \geq \alpha\} \quad (5.14)$$

is called the α -pessimistic value of ζ , where $Tr\{\}$ is the trust measure operator which is defined by (5.12).

Combining (5.14) and (5.12), we have

$$\zeta_{\inf}^{\alpha} = \begin{cases} (1 - 2\alpha)c + 2\alpha d, & 0 < \alpha \leq \frac{a-c}{2(d-c)} \\ 2(1 - \alpha)c + (2\alpha - 1)d, & \frac{b+d-2c}{2(d-c)} \leq \alpha \leq 1 \\ \frac{c(b-a)+a(d-c)+2\alpha(b-a)(d-c)}{(b-a)+(d-c)}, & \frac{a-c}{2(d-c)} < \alpha < \frac{b+d-2c}{2(d-c)} \end{cases} \quad (5.15)$$

We say two rough values $\zeta < \eta$ if and only if, for some predetermined confidence level $\alpha \in (0, 1]$, we have $\zeta_{\inf}^\alpha < \eta_{\inf}^\alpha$.

Definition 4. Let $\zeta = ([a, b], [c, d])$ be a rough variable with $c \leq a \leq b \leq d$, and $\alpha \in (0, 1]$. Then

$$\zeta_{\sup}^\alpha = \sup \{r | Tr\{\zeta \geq r\} \geq \alpha\} \quad (5.16)$$

is called the α -optimistic value to ζ , where $Tr\{\}$ is the trust measure operator which is defined by (5.13).

We say $\zeta < \eta$ if and only if, for some predetermined confidence level $\alpha \in (0, 1]$, we have $\zeta_{\sup}^\alpha < \eta_{\sup}^\alpha$.

Theorem 5. Assume that ζ and η are rough variables. Then, for any $\alpha \in (0, 1]$, we have

1. if $\lambda \geq 0$, then $(\lambda\zeta)_{\inf}^\alpha = \lambda\zeta_{\inf}^\alpha$ and $(\lambda\zeta)_{\sup}^\alpha = \lambda\zeta_{\sup}^\alpha$;
2. if $\lambda < 0$, then $(\lambda\zeta)_{\inf}^\alpha = \lambda\zeta_{\sup}^\alpha$ and $(\lambda\zeta)_{\sup}^\alpha = \lambda\zeta_{\inf}^\alpha$.

Definition 6. Let ζ be a rough variable $([a, b], [c, d])$ with $c \leq a \leq b \leq d$. We then have

$$E[\zeta] = \frac{1}{4}(a + b + c + d)$$

We say $\zeta > \eta$ if and only if $E[\zeta] > E[\eta]$, where E is the expected value operator of rough variables.

5.2.4 Rough Simulation

Rough simulation plays an important role in estimating the value of trust, finding pessimistic or optimistic value, or calculating expected value. The following is an example of estimating the value of trust.

Example 1. Suppose we are given the rough variables $\xi_1 = ([1, 2], [0, 5])$ and $\xi_2 = ([2, 3], [1, 4])$. Let g be a real-valued continuous function given by $g = \xi_1^2 + \xi_2^2$.

In order to obtain the trust $L = Tr\{\xi_1^2 + \xi_2^2 \leq 18\}$, we produce samples λ_1^n and λ_2^n ($n = 1, 2, \dots, N$) from $[0, 5]$ and $[1, 4]$ respectively. Let $\overline{N}$ denote the number of occasions on which $\{\lambda_1^n\}^2 + \{\lambda_2^n\}^2 \leq 18$ for $n = 1, 2, \dots, N$. Then we have the upper trust $\overline{L} = \overline{N}/N$ provided that N is sufficiently large.

A similar way may produce the lower trust $\underline{L}$. We produce samples λ_1^n and λ_2^n ($n = 1, 2, \dots, N$) from $[1, 2]$ and $[2, 3]$ respectively. Let $\underline{N}$ denote the number of occasions on which $\{\lambda_1^n\}^2 + \{\lambda_2^n\}^2 \leq 18$ for $n = 1, 2, \dots, N$. Then we have the lower trust $\underline{L} = \underline{N}/N$.

Thus the trust $L = (\underline{L} + \overline{L})/2$. Rough simulation is performed with $N = 2000$ and obtains $L = 0.82$.

5.3 Rough Program

Rough programs [24] are similar to the constrained mathematical programs except some of the variables involved are rough variables. A constrained mathematical program has the general form given by **MP**.

$$\begin{aligned} \text{MP:} \quad & \min f(X) \\ & \text{subject to} \\ & g_j(X) \leq 0, \quad j = 1, 2, \dots, p \end{aligned} \tag{5.17}$$

where X denotes a vector of decision variables. The objective is to minimize the function $f(X)$ subject to the p constraints which are denoted by $g_j(X) \leq 0$ for $j = 1, \dots, p$. If the objective function or the constraints involve rough variables, denoted by $f(X, \xi)$ and $g_j(X, \xi) \leq 0$ respectively, then **MP** becomes a rough program. There are two basic kinds

of rough programs depending on how the rough variables are ordered. They are known as the expected value rough program and the chance-constrained rough program in [24].

A rough program where rough variables are ordered using their expected values is called an *expected value rough program*. In contrast with the general constrained mathematical program **MP**, the expected value rough program has the general form given by **RPE**.

$$\begin{aligned}
 \mathbf{RPE:} \quad & \min \mathbb{E} [f(X, \xi)] \\
 & \text{subject to} \\
 & \mathbb{E} [g_j(X, \xi)] \leq 0, \quad j = 1, 2, \dots, p
 \end{aligned} \tag{5.18}$$

The expected values only describe the average case. Thus the expected value rough program is not an appropriate choice for our power-balanced VLIW instruction scheduling problem which needs to ensure with a predetermined confidence level that the minimal power variation is obtained.

Another choice is the chance-constrained rough program. Since the constraint functions $g_j(X, \xi)$ are rough variables, they do not produce a deterministic feasible set of decision variables. A natural idea is to provide a predetermined confidence level β at which the rough constraints hold. Thus the constraints become chance constraints having the form

$$Tr\{g_i(X, \xi) \leq 0\} \geq \beta \quad j = 1, 2, \dots, p \tag{5.19}$$

which are defined using a suitable trust measure Tr . In contrast with the general constrained mathematical program **MP**, the chance-constrained rough program has the general form given by **RPC**.

$$\mathbf{RPC:} \quad \min f(X, \xi)_{\inf}^{\alpha}$$

subject to

$$Tr\{g_i(X, \xi) \leq 0\} \geq \beta \quad j = 1, 2, \dots, p \quad (5.20)$$

where the objective function is $f(X, \xi)_{\inf}^\alpha$ instead of $f(X, \xi)_{\sup}^\alpha$, because this formulation is to seek the minimum. The α -pessimistic value is more conservative to guarantee that. Thus it is the appropriate choice to formulate the power-balanced VLIW instruction scheduling problem which needs to ensure with a predetermined confidence level that the minimal power variation is obtained.

In the case that the objective is to seek the maximum, the objective function should be $f(X, \xi)_{\sup}^\alpha$ and the chance-constrained rough program becomes

RPC ':

$$\max f(X, \xi)_{\sup}^\alpha$$

subject to

$$Tr\{g_i(X, \xi) \leq 0\} \geq \beta \quad j = 1, 2, \dots, p \quad (5.21)$$

A hybrid intelligent algorithm which integrates rough simulation, neural network (NN) and genetic algorithm (GA) has been proposed in [24] to solve general rough programs. Because the objective functions in these rough programs are multimodal and the search spaces are irregular, GA can be an appropriate tool to locate good solutions fast. In the process of searching for the optimum, ranking and ordering of the candidate solutions, which involves employing rough simulation to compute their objective function values, are required. As introduced in Section 5.2.4, rough simulation is a computationally expensive process. Therefore this algorithm first generates input-output data for the functions which involve rough variables by rough simulation. The generated input-output data are used to train a neural network to approximate the functions which involve rough variables. Thus the objective function values of the candidate solutions can be more efficiently obtained using the trained neural network. The hybrid intelligent algorithm is

summarized as in Algorithm 3.

- 1 Generate training input-output data for uncertain functions like

$$\begin{aligned} U_1 : X &\rightarrow E[f(X, \xi)], \\ U_2 : X &\rightarrow Tr\{g_j(X, \xi) \leq 0, j = 1, 2, \dots, p\}, \\ U_3 : X &\rightarrow \max\{\bar{f} | Tr\{f(X, \xi) \geq \bar{f}\} \geq \alpha\} \end{aligned}$$

by rough simulations.

- 2 Train a neural network to approximate the uncertain functions according to the generated training data.
- 3 Initialize *pop_size* chromosomes whose feasibility may be checked by the trained neural network.
- 4 Update the chromosomes by crossover and mutation operations in which the feasibility of offsprings may be checked by the trained neural networks.
- 5 Calculate the objective values for all chromosomes by the trained neural networks.
- 6 Compute the fitness of each chromosome according to the objective values.
- 7 Select the chromosomes by spinning the roulette wheel.
- 8 Repeat the fourth to seventh steps for a given number of generations.
- 9 Report the best chromosome as the optimal solution.

Algorithm 3: A hybrid intelligent algorithm to solve general rough programs [24].

However, the method to train a neural network to approximate the functions which involve rough variables is not applicable to our instruction scheduling problem because the decision variables are binary instead of continuous. However, in the process of searching for the optimal instruction schedule, rough simulation still has to be used to compute the objective function values of the candidate schedules. In Chapter 7, we shall propose a rule-based approach to rank the candidate schedules very efficiently. The theory associated with this rule-based approach can be formally justified.

Chapter 6

Instruction-level Power Model Using Rough Variables

The VLIW instruction-level power models described in Chapters 2 and 3 involve a number of approximations. The clustering of instructions described in Section 2.2.1.2 as well as imprecision of power measurements means that the power coefficients are inexact. We shall take this inexactness into account by modelling the power coefficients as rough variables, giving us a rough power model. In this chapter, we address the methodological issues of obtaining the rough variable quantities from the measurement data. The imprecision due to the power measurements and the simplifications and assumptions made in order to cope with the complexity of the VLIW architecture are all characterized in the model coefficients. The simpler power model given by (2.6) will be used in this chapter to illustrate the rough modelling techniques.

The techniques described in this chapter can be applied to any physical measurement based VLIW instruction-level power model to characterize the uncertainty in the model coefficients. For ease of reference, the notations used in this chapter follow those used in Chapter 5.

6.1 Collecting Attribute Data

As mentioned in Section 5.2, the concepts we want to represent are the coefficients of a VLIW power model. Hence the only attribute is “power” (w) which is real-valued. In such a one-attribute approximation space, the universe U is actually the set of real numbers.

The attribute data are collected through power measurements using a setup similar to that described in Section 2.2.1.1. For each coefficient c_k , we conduct measurements with instruction instances in each instruction cluster with different instructions combinations, addressing modes and operands. Furthermore, we also randomly repeat the measurement of an instruction instant a number of times to encapsulate the imprecision inherent in physical measurements. A randomized sequence is necessary to prevent the effects of unknown nuisance variables from contaminating the results. The result of one measurement is treated as an object. Thus the measurement results for all the coefficients constitute the universe U .

6.2 Discretization

The continuous attribute values are then discretized using the Boolean reasoning algorithm described in Section 5.2.1. Let $[l, r] \in \mathfrak{R}$ be the domain of attribute w . The set of cuts obtained by the Boolean reasoning algorithm on $[l, r]$, $B = \{b_1, b_2, \dots, b_m\}$ partition $[l, r]$ into subintervals $[b_0, b_1), [b_1, b_2), \dots, [b_m, b_m + 1]$, where $b_0 = l$ and $b_m + 1 = r$.

6.3 Power Model Coefficients Represented by Rough Variables

Then the original U is transformed into U' by, for $\forall x \in U$, replacing $x = d$ with $x = [b_i, b_{i+1})$ where $d \in [b_i, b_{i+1})$. Then $\forall x, y \in U'$ are indiscernible from each other if they are assigned with the same interval. This indiscernible relation partitions U' into equivalence classes. Let $I(y)$ denote the equivalence class containing y . The rough variable $([a, b], [c, d])$ that represents a power model coefficient c_k is obtained by

$$[a, b] = \cup \{Y_y \mid y \in U' \text{ and } I(y) \subseteq c_k\} \quad (6.1)$$

$$[c, d] = \cup \{Y_y \mid y \in U' \text{ and } I(y) \cap c_k \neq \emptyset\} \quad (6.2)$$

where Y_y denotes the interval y belongs to. Thus all the values in $[a, b]$ are valid description of c_k with certainty and the values in $[c, d]$ are valid description of c_k possibly.

6.4 Rough Power Model of C6711

We use the C6711 [2] as the target VLIW processor to illustrate the proposed rough power modeling method.

For simplicity, the instruction set of the processor is partitioned into four clusters only. They are the memory access cluster, the double-precision floating-point arithmetic function cluster, the single-precision floating-point and fixed-point arithmetic function cluster and the miscellaneous cluster. So there are four power coefficients c_1, c_2, c_3, c_4 in this power model.

The proposed method is independent on the number of the clusters. No matter how many clusters the possible instruction pairs in the instruction set is partitioned into, the proposed method characterizes the uncertainty in the coefficients. An extreme case

Table 6.1: Discretized values of attribute power for the power model coefficients of TMS320C6711.

power	coefficient
[190, 191)	c_4
[191, 192)	c_4
[192, 193)	c_4
[193, 204)	c_3
[193, 204)	c_4
[204, 205)	c_3
[204, 205)	c_4
[205, 207)	c_2
[205, 207)	c_3
[205, 207)	c_4
[207, 208)	c_2
[207, 208)	c_3
[208, 210)	c_2
[208, 210)	c_3
[210, 211)	c_2
[210, 211)	c_3
[211, 213)	c_2
[211, 213)	c_3
[213, 215)	c_2
[215, 216)	c_1
[215, 216)	c_2
[216, 217)	c_1
[216, 217)	c_2
[217, 218)	c_1
[217, 218)	c_2
[218, 219)	c_1
[218, 219)	c_2
[219, 220)	c_1
[219, 220)	c_2
[220, 222)	c_1
[222, 225)	c_1
[225, 227)	c_1
[227, 230]	c_1

Table 6.2: Lower and upper approximations for power coefficients of the example in Section 6.4 ($\times 10\text{mA}$).

c_1	$([22.0, 23.0], [21.5, 23.0])$
c_2	$([21.3, 21.5], [20.5, 22.0])$
c_3	$(\emptyset, [20.2, 21.3])$
c_4	$([19.0, 20.2], [19.0, 20.7])$

is that there is only one instruction pair in each cluster. In that case, we only need to characterize the uncertainty due to physical measurements and ignorance of inter-instruction effects. Thus this example employs a very rough partition for simplicity just to illustrate how to characterize uncertainty in the coefficients.

The concepts we want to represent are the four power model coefficients. The only attribute is “power” which is real-valued. The attribute data are collected through power measurements. For each coefficient, we conduct measurements with instruction instances in each instruction cluster with different inter-instruction effects, addressing modes and operands. Furthermore, we also randomly repeat the measurement of an instruction instant a number of times to encapsulate the imprecision inherent in physical measurements. The result of one measurement is treated as an object and the measurement results for all the coefficients constitute the universe U . The real-valued attribute values are then discretized using the Boolean reasoning algorithm described in Section 5.2.1. This results in a new universe U' where the real-valued attribute values are replaced by the intervals they belong to. After removing the same objects in U' , we obtain the set in Table 6.1. Based on Table 6.1, the lower and upper approximations for each coefficient are obtained according to the definitions (6.1) and (6.2). The results are shown in Table 6.2.

6.5 Summary

It is necessary especially for power critical applications to take into account the imprecisions of the power model so that the schedules obtained have guaranteed optimal power variations to a user specified degree of confidence. Different from the models used for the MIP method where the power model coefficients are given as deterministic values in an average sense, the imprecision of the power model coefficients is handled using rough set techniques. The coefficients are then represented as rough variables which characterize the power model imprecision due to both power measurement and the simplifications made in order to cope with the complexity of the VLIW architecture.

Chapter 7

The Rough Programming Method

The rough power model proposed in Chapter 6 is able to capture the inherent imprecision of its parameters. However, this model cannot be used directly with the MIP approach to power-balanced instruction scheduling discussed in Chapter 4. In this chapter, we formulate the optimization problem as a chance-constrained rough program. The resulting optimal schedule is guaranteed to produce minimal power variation with a specified level of confidence, even considering the uncertainty due to the power model imprecision characterized by the rough power model.

In order to search for the optimal schedule solutions to the chance-constrained rough program, we develop a problem-specific genetic algorithm (GA) and branch-and-bound algorithm respectively. While the GA cannot guarantee optimality, the branch-and-bound algorithm does and thus generally has higher time complexity. In the process of searching for the optimum, ranking and ordering of the candidate schedules, which involves employing rough simulation to compute their objective function values, are required. As introduced in Section 5.2.4, rough simulation is a computationally expensive process. We propose a rule-based approach to rank the candidate schedules very efficiently. The theory associated with this rule-based approach can be fully justified mathematically. Then the problem-specific GA and branch-and-bound algorithm us-

ing the proposed rule-based schedule ranking method is developed to solve the chance-constrained rough program.

The rest of this chapter is organized as follows. In Section 7.1, the power-balanced VLIW instruction scheduling is formulated as a chance-constrained rough program. The problem-specific GA using rough simulation to rank candidate schedules are presented in Section 7.2. The proposed rule-based approach to rank the candidate schedules is presented in Section 7.3. The GA and branch-and-bound algorithm which use the proposed rule-based approach to rank the candidates are presented in Sections 7.4 and 7.5 respectively. Section 7.6 presents the experimental results using the Mediabench and Trimaran benchmarks.

7.1 Chance-Constrained Rough Program

Given the rough power model of the target VLIW processor, the power-balanced instruction scheduling problem can be formulated as a chance-constrained rough program. Rough programs are similar to constrained mathematical programs except some of the variables involved are rough variables. The chance-constrained rough program for n instructions and t time slots is given by **P2**.

$$\mathbf{P2:} \quad \min P(X, \xi)_{\inf}^{\alpha}$$

subject to

$$x_i^j \in \{0, 1\} \quad i = 1, \dots, t; j = 1, \dots, n \quad (7.1)$$

$$X = \bigcup_{i,j:x_i^j=1} \{x_i^j\} \quad i = 1, \dots, t; j = 1, \dots, n \quad (7.2)$$

$$\sum_{i=1}^t x_i^j = 1, \quad j = 1, \dots, n \quad (7.3)$$

$$G(X) \leq 0 \quad (7.4)$$

The decision variables x_i^k are binary. x_i^k equals 1 if instruction k is allocated to time slot i ; otherwise it is 0. Hence the set of non-zero decision variables defines the instruction schedule X as stated in (7.2).

7.1.1 The Objective function

The objective is to minimize power variation throughout the duration of the VLIW program. The total power variation for a schedule X is reflected by the total amount of deviation from the average value M squared. Mathematically, it is given by

$$P(X) = \sum_{i=1}^t (I_i - M)^2 \quad (7.5)$$

where I_i is the power consumption in time slot i which can be obtained through the power model. M is the average power over all t time slots and can be computed by

$$M = \frac{1}{t} \left(\sum_{i=1}^t I_i \right) \quad (7.6)$$

The objective function has to relate the power model variables r_k^i in (2.6) to the decision variables x_i^k . r_k^i is the number of instructions being executed in time slot i which belong to the cluster k . The instructions being executed in time slot i include the ones scheduled to be executed in this time slot and those scheduled earlier but execution is not completed. Hence it can be expressed as

$$r_k^i = \sum_{j=1}^n e_k^j x_i^j + \sum_{j=1}^n \sum_{m=1}^{L^j-1} \Lambda(L^j - m) e_k^j x_{i-m}^j, \quad L^j > 1 \quad (7.7)$$

where $e_k^j = 1$ if instruction j belongs to cluster k and zero otherwise. L^j denotes the number of execution pipeline steps for instruction j . The function $\Lambda(x)$ is defined by

$$\Lambda(x) = \begin{cases} 1, & x \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

where x is an integer. Thus based on the power model (2.6), the power consumed at the i -th time slot is given by

$$I_i = \sum_{k=1}^C c_k r_k^i \quad (7.8)$$

Combining (7.7) and (7.8), we have

$$I_i = \sum_{k=1}^C \sum_{j=1}^n c_k e_k^j x_i^j + \sum_{k=1}^C \sum_{j=1}^n \sum_{m=1}^{L^j-1} \Lambda(L^j - m) c_k e_k^j x_{i-m}^j, \quad L^j > 1 \quad (7.9)$$

Let ξ denote the set of power model coefficients c_k . Since these coefficients are rough variables, the function $P(X, \xi)$ also returns rough values. As mentioned in Section 5.2, rough values can be ordered according to their expected values, α -optimistic values or α -pessimistic values. For our problem, the appropriate one is the more conservative α -pessimistic value because we seek to guarantee minimal power variations. Based on (5.14), we have

$$P(X, \xi)_{\inf}^{\alpha} = \inf\{r | Tr\{P(X, \xi) \leq r\} \geq \alpha\} \quad (7.10)$$

We can see that $P(X, \xi)_{\inf}^{\alpha}$ is actually the minimum value that $P(X, \xi)$ is less than with the confidence level α . Thus the schedule with the minimum $P(X, \xi)_{\inf}^{\alpha}$ can guarantee the minimal power variation with the confidence level α . (7.10) is the objective function we seek to minimize. A rough program like **P1**, where rough variables are ordered using their α -pessimistic values, is called a *chance-constrained rough program*.

Given a schedule X , the value of the objective function (7.10) is obtained through a process called *rough simulation* [24]. Assume that there are a total of N power coefficients. For each power coefficient c_k , randomly take R samples (l_k^q , $q = 1, \dots, R$) from its lower approximation and another R samples (u_k^q , $q = 1, \dots, R$) from its upper approximation for some constant R . The value of $P(X, \xi)_{\inf}^{\alpha}$ is given by the minimum value of v such that

$$\frac{l(v) + u(v)}{2R} \geq \alpha \quad (7.11)$$

where $0 \leq l(v) \leq R$ and $0 \leq u(v) \leq R$ are the number of times $P(X, l_1^q, \dots, l_k^q, \dots, l_N^q)_{\inf}^\alpha \leq v$ and $P(X, u_1^q, \dots, u_k^q, \dots, u_N^q)_{\inf}^\alpha \leq v$ are true as q is increased from 1 to R .

7.1.2 Constraints

Constraint (7.3) ensures that each instruction must be issued once and only once. The performance, data flow dependency and resource usage constraints are given by (7.4). These constraints do not involve any rough variables and are therefore are the same as (4.3)-(4.9) in the MIP formulation **P1** in Section 4.1.2.

7.1.3 Solution

The objective function (7.10) is multimodal and the search space is irregular. Thus genetic algorithms can be an appropriate tool to locate good schedules fast. A genetic algorithm (GA) based algorithm proposed in [24] has been introduced in Section 5.3 to solve general rough program. In the process of searching for the optimum, ranking and ordering of the candidates, which involves rough simulations, are required. In the following section, we propose a problem-specific GA to solve **P2**.

7.2 A Genetic Algorithm Using Rough Simulation to Rank Schedules

Genetic algorithms are population-based iterative adaptive algorithms that are based on natural selection and biological genetics [56]. They encode a potential solution to a specific problem by simple chromosome-like data structures and apply crossover/mutation operators to a population of these structures to achieve optimization. A problem-specific GA is developed to solve **P1**.

7.2.1 Chromosome encoding and initial population

Each schedule is represented as a chromosome. A chromosome is an array of integer variables where an integer variable represents an instruction and its integer value indicates the execution time slot allocated to that instruction. An initial population of feasible candidate schedules is created randomly and may be “seeded” with the optimal schedule obtained through the performance-oriented scheduling algorithms. The size of the population will be denoted by *pop_size*.

7.2.2 Fitness evaluation

The fitness of schedules in each generation is evaluated by the objective function (7.10). The objective function value of each schedule is computed using the rough simulation described in Section 7.1.1. Then the candidate schedules in each generation are sorted in a non-decreasing order by their objective function values.

7.2.3 Genetic operations

The initial population of schedules are then evolved through applying the genetic operations of selection, crossover and mutation until the optimal solution is found.

7.2.3.1 Selection Scheme

The rank-based roulette-wheel selection scheme [56] is adopted for selection. Suppose the candidate schedules of a generation have been sorted in non-decreasing order of fitness. The i -th schedule is assigned a probability of selection by a nonlinear function, $q(i) = a(1 - a)^{i-1}$. The actual selection is performed using the roulette-wheel procedure shown in Algorithm 4.

```

1 for  $i \leftarrow 1$  to  $pop\_size$  do
2     Calculate accumulative probability for the  $i$ -th candidate schedule:  $p_i \leftarrow \sum_{j=1}^i q(j)$ ;
3 end
4 Generate a random number  $r$  within  $[0, p_{pop\_size}]$ ;
5 Select the  $i$ -th candidate schedule such that  $p_{i-1} < r < p_i$ ;

```

Algorithm 4: The rank-based roulette-wheel selection scheme.

7.2.3.2 Crossover and mutation

The selection process is repeated until a population with a size of pop_size is produced. The population of schedules are divided into pairs. Each pair is given a probability, governed by the crossover rate, to be the parents for crossover operation. A 2-point crossover operator is applied. This operator selects two cutting points at random and alternately copies each segment out of the two parent chromosomes. The crossover process may produce unfeasible schedules which violate the constraints on performance, data flow dependency or resource usage as given by (7.4). If infeasible schedules are created, they are not used to replace the parent schedules.

In order to prevent premature convergence, each candidate schedule has a probability to be mutated as governed by the mutation rate. A mutation operator which randomly changes the allocated time slots of an instruction is applied. Similar to crossover, if an infeasible schedule is created, it will not replace the original one.

When a pre-determined number of generations is reached, the algorithm stops. The maximum number of generations depends on the size of the problem, i.e. the number of instructions and the number of available time slots.

Table 7.1: Symbolic power consumption values to illustrate Definition 7.

time slot	...	i	...	j	...
X_1	...	$A + c$	...	B	...
X_2	...	A	...	$B + c$	...

7.3 A Rule-based Approach to Ranking Schedules

During the search for the optimal schedule, candidate schedules X have to be ranked according to their objective function values $P(X, \xi)_{\inf}^{\alpha}$. Computing $P(X, \xi)_{\inf}^{\alpha}$ involves rough simulation which is a computationally expensive process. A much more efficient rule-based approach is developed in this section.

7.3.1 Theoretical Basis

This approach is based on our observation that the difference between the objective function values of two schedules can be obtained by comparing the difference between the two schedules themselves. Definition 7 describes the most basic difference between the power consumption profiles of any two schedules.

Definition 7. *Consider two schedules X_1 and X_2 . If the power consumption of the two schedules are exactly the same at every time slot except i and j as shown in Table 7.1, where a rough power model variable c is moved from time slot i to j , then we say that X_2 exhibits a ‘OneMove’ from X_1 .*

Our hypothesis that for two schedules X_1 and X_2 , the difference $P(X_2, \xi)_{\inf}^{\alpha} - P(X_1, \xi)_{\inf}^{\alpha}$ is dependent on the α -pessimistic values of the power variables in time slots i and j only. If this hypothesis is true, the α -pessimistic value operator should be linear. Lemma 8 proves this linearity property when α is large enough. Based on Lemma 8, Theorem 9 proves our hypothesis.

Lemma 8. Given two rough variables $y = ([a_y, b_y], [c_y, d_y])$ and $z = ([a_z, b_z], [c_z, d_z])$, let $u = y + z$. Then

$$u_{\inf}^{\alpha} = y_{\inf}^{\alpha} + z_{\inf}^{\alpha} \quad (7.12)$$

provided $\alpha \in (0, 1]$ satisfies

$$\alpha \geq \max \left(\frac{b_y + d_y - 2c_y}{2(d_y - c_y)}, \frac{b_z + d_z - 2c_z}{2(d_z - c_z)} \right) \quad (7.13)$$

Proof. Since u is the sum of y and z , the lower and upper approximations of u are computed by adding the values of the corresponding limits (Refer to Section 5.2.2 for rough arithmetics).

$$u = ([a_y + a_z, b_y + b_z], [c_y + c_z, d_y + d_z])$$

Let $u_{\inf}^{\alpha}$, $y_{\inf}^{\alpha}$ and $z_{\inf}^{\alpha}$ be the α -pessimistic values of u , y and z respectively. Based on (5.15), these values are given by

$$u_{\inf}^{\alpha} = \begin{cases} (1 - 2\alpha)(c_y + c_z) + 2\alpha(d_y + d_z), & 0 < \alpha \leq \frac{a_y + a_z - c_y - c_z}{2(d_y + d_z - c_y - c_z)} \\ 2(1 - \alpha)(c_y + c_z) + (2\alpha - 1)(d_y + d_z), & p_u \leq \alpha \leq 1 \\ \frac{(c_y + c_z)(b_y + b_z - a_y - a_z) + (a_y + a_z)(d_y + d_z - c_y - c_z)}{(b_y + b_z - a_y - a_z) + (d_y + d_z - c_y - c_z)} + \frac{2\alpha(b_y + b_z - a_y - a_z)(d_y + d_z - c_y - c_z)}{(b_y + b_z - a_y - a_z) + (d_y + d_z - c_y - c_z)}, & \text{otherwise} \end{cases}$$

$$y_{\inf}^{\alpha} = \begin{cases} (1 - 2\alpha)c_y + 2\alpha d_y, & 0 < \alpha \leq \frac{a_y - c_y}{2(d_y - c_y)} \\ 2(1 - \alpha)c_y + (2\alpha - 1)d_y, & p_y \leq \alpha \leq 1 \\ \frac{c_y(b_y - a_y) + a_y(d_y - c_y) + 2\alpha(b_y - a_y)(d_y - c_y)}{(b_y - a_y) + (d_y - c_y)}, & \text{otherwise} \end{cases} \quad (7.14)$$

$$z_{\inf}^{\alpha} = \begin{cases} (1 - 2\alpha)c_z + 2\alpha d_z, & 0 < \alpha \leq \frac{a_z - c_z}{2(d_z - c_z)} \\ 2(1 - \alpha)c_z + (2\alpha - 1)d_z, & p_z \leq \alpha \leq 1 \\ \frac{c_z(b_z - a_z) + a_z(d_z - c_z) + 2\alpha(b_z - a_z)(d_z - c_z)}{(b_z - a_z) + (d_z - c_z)}, & \text{otherwise} \end{cases} \quad (7.15)$$

where

$$\begin{aligned} p_u &= \frac{b_y + b_z + d_y + d_z - 2(c_y + c_z)}{2(d_y + d_z - c_y - c_z)} \\ p_y &= \frac{b_y + d_y - 2c_y}{2(d_y - c_y)} \\ p_z &= \frac{b_z + d_z - 2c_z}{2(d_z - c_z)} \end{aligned}$$

Note that $p_u < \max(p_y, p_z)$.

Hence when $\alpha \geq \max(p_y, p_z)$, we have $\alpha \geq p_u$. In this case, we have

$$u_{\inf}^\alpha = y_{\inf}^\alpha + z_{\inf}^\alpha$$

This completes the proof. □

Our optimization problem is to guarantee the minimal power variation with the confidence level α . Thus a large enough confidence level is needed in which case $\alpha \geq \max(p_y, p_z)$ is generally required. In the rest of this thesis, we assume that α satisfies this condition.

Theorem 9. *Suppose schedule X_2 exhibits a 'OneMove' from schedule X_1 in time slots i and j as shown in Table 7.1. The other power variables in time slots i and j are respectively denoted as $A = a_1 + a_2 + \dots + a_n$ and $B = b_1 + b_2 + \dots + b_m$. If α satisfies the condition stated in (7.13), we have:*

$$P(X_2, \xi)_{\inf}^\alpha - P(X_1, \xi)_{\inf}^\alpha = \sum_{g=1}^m (2b_g c)_{\inf}^\alpha - \sum_{g=1}^n (2a_g c)_{\inf}^\alpha \quad (7.16)$$

Proof. The power variation defined by (7.5) for a given schedule can be expressed as

$$\begin{aligned} P(X_1, \xi) &= \left(\sum_{k=1, k \neq i, k \neq j}^t (P_k - M)^2 \right) + (P_i - M)^2 + (P_j - M)^2 \\ P(X_2, \xi) &= \left(\sum_{k=1, k \neq i, k \neq j}^t (P_k - M)^2 \right) + (P_i - M)^2 + (P_j - M)^2 \end{aligned}$$

Hence according to Lemma 8, with α large enough, we have

$$\begin{aligned}
 P(X_1, \xi)_{\inf}^\alpha &= \left(\sum_{k=1, k \neq i, k \neq j}^t (P_k - M)^2 \right)_{\inf}^\alpha + ((P_i - M)^2 + (P_j - M)^2)_{\inf}^\alpha \\
 &= \left(\sum_{k=1, k \neq i, k \neq j}^t (P_k - M)^2 \right)_{\inf}^\alpha + ((A + c - M)^2 + (B - M)^2)_{\inf}^\alpha \\
 P(X_2, \xi)_{\inf}^\alpha &= \left(\sum_{k=1, k \neq i, k \neq j}^t (P_k - M)^2 \right)_{\inf}^\alpha + ((P_i - M)^2 + (P_j - M)^2)_{\inf}^\alpha \\
 &= \left(\sum_{k=1, k \neq i, k \neq j}^t (P_k - M)^2 \right)_{\inf}^\alpha + ((A - M)^2 + (B + c - M)^2)_{\inf}^\alpha
 \end{aligned}$$

The difference is given by

$$\begin{aligned}
 &P(X_2, \xi)_{\inf}^\alpha - P(X_1, \xi)_{\inf}^\alpha \\
 &= ((A - M)^2 + (B + c - M)^2)_{\inf}^\alpha - ((A + c - M)^2 + (B - M)^2)_{\inf}^\alpha \\
 &= (2Bc)_{\inf}^\alpha - (2Ac)_{\inf}^\alpha \\
 &= \sum_{g=1}^m (2b_g c)_{\inf}^\alpha - \sum_{g=1}^n (2a_g c)_{\inf}^\alpha
 \end{aligned}$$

Hence, this theorem has been proved. $\square$

Corollary 10. *Suppose the schedule X_2 exhibits K ‘OneMove’ from the schedule X_1 with $K > 1$. Then the difference between the objective function values of X_2 and X_1 is equal to the sum of the differences caused by each of the K ‘OneMove’.*

Proof. First consider $K = 2$. We construct an intermediate schedule X_3 which exhibits one ‘OneMove’ from X_1 and another ‘OneMove’ to X_2 . Then

$$\begin{aligned}
 &P(X_2, \xi)_{\inf}^\alpha - P(X_1, \xi)_{\inf}^\alpha \\
 &= P(X_2, \xi)_{\inf}^\alpha - P(X_1, \xi)_{\inf}^\alpha + P(X_3, \xi)_{\inf}^\alpha - P(X_3, \xi)_{\inf}^\alpha \\
 &= [P(X_2, \xi)_{\inf}^\alpha - P(X_3, \xi)_{\inf}^\alpha] + [P(X_3, \xi)_{\inf}^\alpha - P(X_1, \xi)_{\inf}^\alpha]
 \end{aligned}$$

This completes the proof for $K = 2$. $K > 2$ can be proved similarly. $\square$

Theorem 9 tells us that the difference between the objective function values of two schedules is only dependent on the α -pessimistic values of the products of power variables in the time slots which exhibit ‘OneMove’ differences. Therefore if we pre-compute all possible combinations of the power model coefficients, then any rough simulation results could be obtained from these values. They can be expressed as rules where the premise is a pair of power model coefficients and the conclusion is the α -pessimistic value of the product of this combination obtained by rough simulation. Suppose there are a total of C coefficients in the rough power model. Then there will be $\frac{1}{2}(C^2 + C)$ rules in total.

This rule extraction process is done offline only once for the rough power model of a target VLIW processor. Then by comparing the ‘OneMove’ differences between two schedules, the difference between their objective function values can be obtained using these rules.

7.3.2 Examples

We shall continue with the rough power model given in the example in Section 6.4. In the following examples, the power coefficient rules are established and they are used to compute the difference of objective function values of two given schedules.

Example 2. *Consider the rough power model in the example in Section 6.4. The premise of the rules are the combinations of any two coefficients among c_1 , c_2 , c_3 and c_4 . So there are a total of ten rules. Let $\alpha = 0.95$. We compute the α -pessimistic values of the products of the combinations using rough simulation. The ten rules are summarized as in Table 7.2.*

Example 3. *There are seven instructions to be scheduled in five time slots and two feasible schedules X_1 and X_2 are shown in Table 7.3. Suppose the instructions s_1 , s_2 , s_3 and s_7 belong to cluster c_4 and the instructions s_4 , s_5 and s_6 belong to cluster c_1 . Then the symbolic power consumption profiles of X_1 and X_2 are as shown in the table.*

Table 7.2: Power difference rules for Example 2.

1	c_1	c_1	1063.46
2	c_1	c_2	1006.48
3	c_1	c_3	976.78
4	c_1	c_4	945.87
5	c_2	c_2	952.26
6	c_2	c_3	924.00
7	c_2	c_4	894.96
8	c_3	c_3	896.67
9	c_3	c_4	868.64
10	c_4	c_4	841.52

Table 7.3: Two instruction Schedules used in Example 3.

X_1	TimeSlot	1	2	3	4	5
	Instructions	s_1	s_2, s_3	s_4	s_5, s_6	s_7
	Power	c_1	$c_1 + c_1$	c_2	$c_2 + c_2$	c_1
X_2	TimeSlot	1	2	3	4	5
	Instructions	s_1	s_2	s_3, s_4	s_5	s_6, s_7
	Power	c_1	c_1	$c_1 + c_2$	c_2	$c_1 + c_2$

X_2 exhibits a difference of two 'OneMove' from X_1 – one is from time slot 2 to 3 and the other is from time slot 4 to 5. Thus according to Corollary 10 and Theorem 9, the difference between the objective function values of X_2 and X_1 is given by

$$P(X_2, \xi)_{\inf}^{\alpha} - P(X_1, \xi)_{\inf}^{\alpha} = (2c_4c_1)_{\inf}^{\alpha} - (2c_1c_1)_{\inf}^{\alpha} + (2c_4c_1)_{\inf}^{\alpha} - (2c_4c_4)_{\inf}^{\alpha}$$

The four α -pessimistic values can be obtained in Table 7.2. Thus

$$P(X_2, \xi)_{\inf}^{\alpha} - P(X_1, \xi)_{\inf}^{\alpha} = 2 \times 853.0 - 987.9 - 744.1 = -26$$

This result shows that X_2 is better than X_1 , namely X_2 can guarantee lower power variation than X_1 with the confidence level 0.95, even considering the uncertainty due to the power model imprecision characterized by the rough power model.

7.4 A Genetic Algorithm with Rule-Based Schedule Ranking

The genetic algorithm using rule-based schedule ranking is similar with the genetic algorithm using rough simulation to rank schedules which has been presented in Section 7.2. The only difference is the methods used for fitness evaluation. In both algorithms, the fitness of schedules in each generation is evaluated by the objective function (7.10). However, the rule-based approach can obtain the difference between the objective function values of any two candidate schedules. Then the candidate schedules in each generation are sorted in a non-decreasing order by their objective function values. This is much more efficient than using rough simulation to compute the objective function values for each candidate schedule respectively as in the genetic algorithm described in Section 7.2.

In order to obtain the objective value of the best schedule obtained, rough simulation has to be conducted, because the rule-based approach only computes the differences between the objective function values during the search.

7.5 Branch-and-Bound Algorithm with Rule-Based Schedule Ranking

The branch-and-bound algorithm developed to solve **P2** in this section can guarantee optimality while the GA presented in Section 7.2 cannot. Compare to the MIP problem formulation **P1** described in Section 4.1.2, the rough program formulation **P2** only has a different objective function. Thus the branch-and-bound algorithm to solve **P2** can be based on the branch-and-bound algorithm to solve **P1** which has been presented in Section 4.2. However, some modifications are necessary to the power model related points in the branch-and-bound algorithm to solve **P1**.

7.5.1 Preliminaries

First, we need to pre-compute the rough simulation results for the α -pessimistic values of each single power model coefficient c_k .

Second, using the rough power model, the formula for calculating I_i^u (4.22) needs to be changed. The formula for calculating I_i^u can be obtained based on (7.9). It is given by

$$I_i^u = \sum_{k=1}^C \sum_{\forall j \in U_r} c_k e_k^j x_i^j + \sum_{k=1}^C \sum_{\forall j \in U_r} \sum_{m=1}^{L^j-1} \Lambda(L^j - m) c_k e_k^j x_{i-m}^j, \quad L^j > 1 \quad (7.17)$$

7.5.2 Branching Rules

First, the rule-based schedule ranking method is used to compare the feasible schedules branching from the selected leaf schedule X_r and the current best schedule X_{CBS} . If the obtained differences between their objective function values show any feasible schedule X_f is better than X_{CBS} , X_f becomes the new X_{CBS} .

Second, the equation (4.23) in Condition 1 given in Section 4.2.2 should be changed

to rank them with their α -pessimistic values:

$$\{I_j^u\}_{\inf}^\alpha \leq P_{\max}(X_{CBS}) \quad (7.18)$$

where $\{I_j^u\}_{\inf}^\alpha$ is computed according to Lemma 8 and Theorem 5 as

$$\begin{aligned} \{I_j^u\}_{\inf}^\alpha &= \sum_{k=1}^C \sum_{\forall j \in U_r} \{c_k e_k^j x_i^j\}_{\inf}^\alpha + \sum_{k=1}^C \sum_{\forall j \in U_r} \sum_{m=1}^{L^j-1} \{\Lambda(L^j - m) c_k e_k^j x_{i-m}^j\}_{\inf}^\alpha \\ &= \sum_{k=1}^C \sum_{\forall j \in U_r} e_k^j x_i^j \{c_k\}_{\inf}^\alpha + \sum_{k=1}^C \sum_{\forall j \in U_r} \sum_{m=1}^{L^j-1} \Lambda(L^j - m) e_k^j x_{i-m}^j \{c_k\}_{\inf}^\alpha \end{aligned} \quad (7.19)$$

for $L^j > 1$. The α -pessimistic value of I_i defined by (7.9) can also be computed in a similar way

$$\{I_i\}_{\inf}^\alpha = \sum_{k=1}^C \sum_{j=1}^n e_k^j x_i^j \{c_k\}_{\inf}^\alpha + \sum_{k=1}^C \sum_{j=1}^n \sum_{m=1}^{L^j-1} \Lambda(L^j - m) e_k^j x_{i-m}^j \{c_k\}_{\inf}^\alpha \quad (7.20)$$

for $L^j > 1$. Then $P_{\max}(X_{CBS})$ is the largest $\{I_i\}_{\inf}^\alpha$ ($i = 1, \dots, t$) in the current best schedule.

7.5.3 Lower Bound Estimation

Similar as the lower bound estimation approach presented in Section 4.2.4, we also redistribute the power consumption of instructions in V_r in order to obtain the minimum of the objective function (7.10), which is a lower bound of the objective function values for the successors of X_r .

The total power consumption of each instruction in V_r is composed of power consumption spread out over all its execution time slots. For an instruction in the k -th cluster where $k = 1, \dots, C$, the rough power model coefficient c_k is then the average power consumed by it *per time slot*, assuming uniform distribution. Let y_i^k ($k = 1, \dots, C$) denote the number of units of c_k ($k = 1, \dots, C$) in time slot i respectively. Then we can formulate an alternative linear program, $\mathbf{P}_{\mathbf{LB}}$, based on these variables. The optimal solution of

$\mathbf{P}_{\text{LB}}$ will provide us with a lower bound of the objective function values for the successors of X_r .

$$\mathbf{P}_{\text{LB}}: \quad \min P(X, \xi)_{\text{inf}}^{\alpha}$$

subject to

$$y_i^k \geq 0, \quad k = 1, \dots, C \quad (7.21)$$

$$\text{Tr} \left\{ I_i^v = \sum_{k=1}^C y_i^k c_k \right\} \geq \alpha, \quad i = 1, \dots, t \quad (7.22)$$

$$\text{Tr} \left\{ \sum_{i=1}^t I_i^v = \sum_{\forall j \in V_r} \sum_{k=1}^C e_k^j c_k D^j \right\} \geq \alpha \quad (7.23)$$

where D^j denotes the number of execution time slots of instruction j and $e_k^j = 1$ if instruction j belongs to cluster k and zero otherwise. In the objective function, $P(X, \xi)$ is computed using the following equation

$$P(X, \xi) = \sum_{i=1}^t (I_i - M)^2 \quad (7.24)$$

where

$$I_i = I_i^u + I_i^v, \quad i = 1, \dots, t \quad (7.25)$$

$$I_i^v = \sum_{k=1}^C y_i^k c_k, \quad i = 1, \dots, t \quad (7.26)$$

$$M = \frac{1}{t} \left(\sum_{i=1}^t I_i \right) \quad (7.27)$$

Theorem 11. *For a given schedule X_r , the optimal value of $P(X, \xi)_{\text{inf}}^{\alpha}$ obtained by the chance-constrained rough program $\mathbf{P}_{\text{LB}}$ is a lower bound of the objective function $P(X, \xi)_{\text{inf}}^{\alpha}$ of the rough program $\mathbf{P2}$ for all successors of the schedule X_r .*

Proof. For a given schedule X_r , let s_{LB} denote the optimal value of $P(X, \xi)_{\text{inf}}^{\alpha}$ for the program $\mathbf{P}_{\text{LB}}$. We need to prove that for any successor X_s of X_r , $s_{\text{LB}} \leq P(X_s, \xi)_{\text{inf}}^{\alpha}$.

Let $\mathbf{P2}'$ be the subproblem of $\mathbf{P2}$ with the partial schedule X_r . It is the rough program with binary decision variables x_i^k for $k \in V_r$ and $i = 1, \dots, t$. Let $s_{P2'}$ denote the optimal objective function value of $\mathbf{P2}'$. Then for any successor X_s of X_r ,

$$s_{P2'} \leq P(X_s, \xi)_{\inf}^{\alpha} \quad (7.28)$$

Now compare $\mathbf{P_{LB}}$ with $\mathbf{P2}'$ for a given schedule X_r . The two formulations have the same objective function. In constraints (7.22) and (7.23), the trust measure operations are only used to guarantee the two equations with the confidence level α since rough variables are involved. But in $\mathbf{P_{LB}}$, the data dependency and resource constraints on the instructions in V_r have been removed. Since $\mathbf{P_{LB}}$ is actually the same problem as $\mathbf{P2}'$ but with less constraints, the optimal value of $\mathbf{P_{LB}}$ must be less than the one obtained by $\mathbf{P2}'$. That is, for the given schedule X_r ,

$$s_{LB} \leq s_{P2'} \quad (7.29)$$

Therefore, based on (7.28) and (7.29), we have $s_{LB} \leq P(X_s, \xi)_{\inf}^{\alpha}$ for any successor X_s of X_r . $\square$

One way to guarantee the trust of the two equations in constraints (7.22) and (7.23) is to let the two sides of an equation have an equal α -pessimistic value. Hence constraints (7.22) and (7.23) can be substituted by

$$\{I_i^v\}_{\inf}^{\alpha} = \sum_{k=1}^C y_i^k \{c_k\}_{\inf}^{\alpha} \quad (7.30)$$

$$\sum_{i=1}^t \{I_i^v\}_{\inf}^{\alpha} = \sum_{\forall j \in V_r} \sum_{k=1}^C e_k^j D^j \{c_k\}_{\inf}^{\alpha} \quad (7.31)$$

Hence in order to solve $\mathbf{P_{LB}}$, the optimal power consumption profile represented with the power coefficients can be obtained by using a simple water-filling algorithm as shown in Figure 5. This algorithm starts with the schedule where each time slot already has

power consumption described by I_i^u . The power units c_k of the instructions in V_r are filled into the time slots of this schedule by filling the time slots with lowest α -pessimistic value of the current power consumption first. Those power units c_k which have larger α -pessimistic value are chosen first because it is easier to use the smaller portions to “fill in the gaps” later so that power variations are minimized.

We need to compare the obtained lower bound estimation for a given leaf schedule X_r with the objective function value of the current best schedule. The difference can be obtained using the rule-based schedule ranking method, by finding out the ‘OneMove’s between the optimal power consumption profile obtained by Algorithm 5 and the power consumption profile of the current best schedule.

input : The chance-constrained rough program $\mathbf{P}_{\text{LB}}$ for lower bound estimation of the schedule X_r .

output: the optimal power consumption profile represented with the power coefficients.

- 1 calculate each $\{I_i^u\}_{\text{inf}}^\alpha$ according to (7.17) and (7.19); initiate each $\{I_i\}_{\text{inf}}^\alpha$ with its $\{I_i^u\}_{\text{inf}}^\alpha$: $\{I_i\}_{\text{inf}}^\alpha = \{I_i^u\}_{\text{inf}}^\alpha$;
- 2 tempCount $\leftarrow$ number of units of the largest $\{c_k\}_{\text{inf}}^\alpha$ consumed by the instructions in V_r ;
- 3 **for** $i \leftarrow 1$ **to** tempCount **do**
- 4 Suppose the value in time slot W is the smallest among the current $\{I_i\}_{\text{inf}}^\alpha$ ($i = 1$ to t), then $\{I_W\}_{\text{inf}}^\alpha = \{I_W\}_{\text{inf}}^\alpha + \{c_k\}_{\text{inf}}^\alpha$;
- 5 **end**
- 6 **if** All the units $\{c_k\}_{\text{inf}}^\alpha$ consumed by the instructions in V_r have been distributed **then** Stop. ;
- 7 **else** tempCount $\leftarrow$ the number of units of the largest $\{c_k\}_{\text{inf}}^\alpha$ in the remain units consumed by the instructions in V_r ; Go to Step 4

Algorithm 5: A water-filling algorithm to obtain the optimal power consumption profile to the rough program $\mathbf{P}_{\text{LB}}$.

7.6 Performance Evaluation

We use the C6711 and the Trimaran [26] and MediaBench [25] benchmarks for performance evaluation. The rough power model of C6711 and its rules for schedule ranking have been presented in Section 6.4 and Example 2 separately. For each problem instance extracted from these benchmarks, a speed optimized schedule is obtained using a performance-oriented compiler. This schedule provides the number of time slots available for scheduling. So there is no performance degradation for the power-balanced schedules obtained. In practice, if the execution performance of the target program can be relaxed, then a larger t can be used. The VLIW instructions are scheduled by the rough programming method. Power-balanced schedules are also obtained through mixed-integer programming as discussed in Chapter 4 using a deterministic power model.

In our problem, the crossover operator often cannot create enough feasible offsprings due to violations of the constraints. Therefore, in order to prevent premature convergence, a low crossover rate of 0.2 and a high mutation rate of 0.8 are used. The population size and generation are 30 and 30 respectively. Tuning these parameter values to be suitable for a specific data set may give rise to improved performance of the GA.

For any problem instance, we schedule instructions using the mixed-integer programming method in Chapter 4 and this rough programming method separately. The schedules obtained by rough programming are compared with those obtained by mixed-integer programming, with respect to their values of the objective function defined by (7.10). Next, we give a simple example to illustrate this comparison.

Example 4. Consider a problem instance consisting of fourteen instructions to be scheduled in six time slots. The instructions are $(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}, S_{11}, S_{12}, S_{13}, S_{14})$ with their data dependency as in Figure 7.1. Suppose the instructions S_5, S_8, S_{12} and S_{13} belong to cluster c_1 , the instructions S_9 and S_{14} belong cluster c_4 , and the

Figure 7.1: Data dependency graph of instructions in Example 4.

others belong to c_3 . Suppose there is no resource usage constraints imposed by the target processor.

The schedule obtained by performance-oriented compiler is $X = \{x_1^1, x_2^1, x_3^1, x_4^1, x_5^2, x_6^2, x_7^2, x_8^3, x_9^4, x_{10}^5, x_{11}^5, x_{12}^5, x_{13}^6, x_{14}^6\}$, where the superscripts indicate the time slot where the instruction is being scheduled.

Using the mixed-integer programming method, the optimal schedule is given by $X_{MIP} = \{x_1^1, x_2^1, x_3^1, x_4^1, x_5^2, x_6^2, x_7^2, x_8^3, x_9^4, x_{10}^5, x_{11}^5, x_{12}^5, x_{13}^6, x_{14}^6\}$.

Using the rough programming method, we obtain a schedule $X_{RP} = \{x_1^1, x_2^1, x_3^1, x_4^1, x_5^2, x_6^2, x_7^2, x_8^3, x_9^4, x_{10}^5, x_{11}^5, x_{12}^5, x_{13}^6, x_{14}^6\}$

Then we compare the values of the objective function (7.10) of the three schedules X , X_{MIP} and X_{RP} ,

$$P(X, \xi)_{inf}^{0.95} = 6.6808 \times 10^4 \quad (7.32)$$

$$P(X_{MIP}, \xi)_{inf}^{0.95} = 5.5633 \times 10^4 \quad (7.33)$$

$$P(X_{RP}, \xi)_{inf}^{0.95} = 5.4781 \times 10^4 \quad (7.34)$$

(7.34) means, with confidence level 0.95, the values of $P(X_{RP}, \xi)$ are guaranteed to be less than 5.4781×10^4 even considering the uncertainty due to the imprecision in the

Table 7.4: Experimental results on instruction schedules of various sizes from Trimaran’s benchmark program.

Dim.	Source	X ($\times 10^4$)	X_{MIP} ($\times 10^4$)	X_{RP}^{GA} ($\times 10^4$)	T^{GA} (sec.)	X_{RP}^{BB} ($\times 10^4$)	T^{BB} (sec.)	$Imp.^{GA}$ (%)	$Imp.^{BB}$ (%)
(15,16)	bmm	15.22	12.85	8.43	0.015	8.28	0.109	34.40	35.56
(11,17)	fir	29.56	29.56	10.41	0.016	10.40	0.046	64.78	64.82
(14,10)	mm	5.06	3.80	3.09	0.015	3.07	0.015	18.68	19.21
(9,12)	strcpy	11.87	6.15	4.94	0.016	4.73	0.015	19.67	23.09
(13,13)	wave	8.35	6.84	5.07	0.015	4.86	0.046	25.88	28.95
Average								32.68	34.33

coefficients. (7.33) means, with confidence level 0.95, the values of $P(X_{MIP}, \xi)$ are only less than 5.5633×10^4 considering the imprecision in the coefficients. And the same explanation is with (7.32). We can see that the schedule obtained by rough programming guarantees much smaller power variation compared than the performance optimal schedule and the optimal schedule obtained by integer programming.

Suppose we can get a precise power model exactly, which means that there is only one instruction pair in each cluster and no imprecision inherent in physical measurements, etc. In this case, both methods would return the same optimal schedules. The more uncertainties involved in the power models, the more obvious the superiority of the rough programming method.

The results based on problem instances extracted from Trimaran benchmarks and MediaBench are presented in Tables 7.4 and 7.5 respectively. For each problem instance, the problem dimension (Dim.) is an ordered pair indicating the number of available time slots and the number of instructions.

Columns ' X ', ' X_{MIP} ', ' X_{RP}^{GA} ' and ' X_{RP}^{BB} ' show the values of (7.10) for schedules obtained through the performance-oriented compiler, the mixed-integer program instruction scheduler, the genetic algorithm based rough program scheduler and the branch-and-

Table 7.5: Experimental results on instruction schedules of various sizes from Mediabench program.

Dim.	Source	X ($\times 10^4$)	X_{MIP} ($\times 10^4$)	X_{RP}^{GA} ($\times 10^4$)	T^{GA} (sec.)	X_{RP}^{BB} ($\times 10^4$)	T^{BB} (sec.)	$Imp.^{GA}$ (%)	$Imp.^{BB}$ (%)
(55,34)	adpcm	74.39	65.07	61.68	0.031	55.37	2.467	5.21	14.91
(55,39)	adpcm	94.65	89.18	79.17	0.046	78.77	2.875	11.22	11.67
(28,30)	epic	139.27	130.63	124.71	0.031	124.59	1.250	4.53	4.62
(6,13)	epic	6.82	5.32	4.74	0.015	4.74	0.010	10.90	10.90
(20,15)	epic	11.06	9.81	6.15	0.015	6.15	0.156	37.31	37.31
(19,20)	g721	57.88	57.88	27.20	0.015	27.20	0.156	53.01	53.01
(37,30)	g721	83.25	83.25	58.40	0.046	58.34	2.234	29.85	29.92
(44,23)	gsm	34.70	27.53	19.28	0.046	18.73	2.190	29.97	31.97
(73,46)	gsm	94.48	82.88	55.01	0.046	54.73	3.843	33.63	33.96
(38,34)	jpeg	101.11	101.11	59.55	0.046	55.61	2.342	41.10	45.00
(30,27)	jpeg	48.46	48.46	41.47	0.031	37.97	1.225	14.42	21.65
(55,58)	jpeg	197.70	197.70	162.20	0.062	162.04	2.843	17.96	18.04
Average								24.09	26.08

bound algorithm based rough scheduler respectively. The results show that by taking the power model coefficient imprecisions into account, the rough programming approach is able to produce schedules with smaller power variations. Take the problem instance (55, 34) as an example, $P(X_{RP}^{GA}, \xi)$ is guaranteed to be less than 61.68×10^4 with a confidence level of 0.95. In contrast, $P(X_{MIP}, \xi)$ and $P(X, \xi)$ are only less than 65.07×10^4 and 74.39×10^4 with the same confidence level.

Overall, our results of rough programming solved using genetic algorithm obtain an average improvement 32.68% (Trimaran benchmarks) and 24.09% (Mediabench) of X_{RP}^{GA} related to X_{MIP} . Our results of rough programming solved using branch and bound algorithm obtain an average improvement 34.33% (Trimaran benchmarks) and 26.08% (Mediabench) of X_{RP}^{BB} related to X_{MIP} .

The computation times required by the proposed problem-specific GA (Column ' T^{GA} ') show that it is efficient enough for implementation in production compilers. Although the rule-based branch-and-bound algorithm has relatively longer computation time (Column

Table 7.6: Computation time comparison of the GAs on problem instances with various sizes extracted from Mediabench.

Dim	Source	Rough Simulation GA (sec.)	Rule-based GA (sec.)
(28,30)	epic	101.233	0.031
(37,30)	g721	131.92	0.046
(44,23)	gsm	123.56	0.046
(38,34)	jpeg	152.513	0.046
(55,58)	jpeg	354.636	0.062

T^{BB}), it can guarantee optimality and be used as a reference point for the GA solution. All our computational experiments are performed on a 2.8GHz Intel Pentium 4 personal computer with 512MB RAM running under Microsoft Windows 2000.

We also conducted experiments to show the computational advantage of the rule-based schedule ranking approach by comparing the rule-based genetic algorithm with the genetic algorithm using rough simulation. Table 7.6 shows the comparison results. We perform power-balanced instruction scheduling on five instruction schedules taken from MediaBench. The sample size for rough simulation is set as 50. The crossover probability, mutation rate, population size and generations are the same in both cases. The results show a significant reduction in computation time.

7.7 Summary

A rough programming approach to the power-balanced VLIW instruction scheduling optimization problem has been proposed. This approach takes into account the imprecisions of the power model so that the schedules obtained guarantee to produce optimal power variations to a user specified degree of confidence. The optimization problem is formulated as a chance-constrained rough program. A problem-specific GA and branch-and-bound algorithm are respectively developed to find solutions to the rough program. A computationally efficient rule-based approach is proposed to rank the candidate schedules

without having to use rough simulation which is computationally expensive. Performance evaluation is conducted using the C6711 with MediaBench and Trimaran benchmarks. Our results show that the optimal schedules obtained by our new method can achieve guaranteed respectively 24% and 32% less power variation than the power-balanced schedules obtained by the mixed-integer program approach. The computational requirement of the proposed GA is also shown to be low enough for use in production compilers.

Chapter 8

Conclusions

Although VLIW is an energy-efficient architecture and VLIW instruction scheduling techniques for performance optimizations are adaptable to total energy optimization, instruction schedules that are optimized for speed often exhibit large variation in processor power consumption throughout the execution of the target program. This thesis has made original contributions to the minimization of this power variation by rescheduling instructions without compromising on the execution speed. In this respect, two approaches have been used. The first one is the mixed-integer programming approach and the other approach is based on rough set theory and rough programming.

The optimization problem is first formulated as a mixed-integer program (MIP). While this approach has been proposed before, our MIP problem formulation makes use of a more complete resource usage model. It takes into account constraints imposed by multi-cycle functional unit latency as well as limited shared resources such as registers, read/write ports and buses which have not been considered before. This MIP also makes use of a more accurate VLIW instruction-level power model which is based on the power model in [4]. We improve the original one in [4] in order to be applied to power-balanced instruction scheduling. We have improved the original functional-level concurrent activity analysis which leads to a more accurate power model. We also im-

prove it to allows us to perform fine-grained run-time power profiling by accounting for the overlapping of instructions within the pipeline while the original model only allows us to compute the total power consumed. Finally an efficient branch-and-bound algorithm is developed to solve the MIP much more efficiently than generic solvers, making the technique more attractive for use in practical compilers. Generic MIP solvers do not have problem-specific knowledge to reduce the search space and therefore are inferior to the branch-and-bound algorithm which incorporates such knowledge specifically for this problem so that optimal solutions can be found efficiently.

Experimental results based on the C6711 using benchmarks programs from Media-bench and Trimaran showed that over 40% average reduction in power variation can be achieved at a relatively low computational cost and the execution performance of the target programs is not degraded.

While the optimal schedules obtained using the MIP method are only power balanced in an average sense, the proposed rough programming method does take into account the imprecision of the power model so that the obtained optimal schedule is guaranteed to produce the minimal power variation to a user specified degree of confidence. In contrast to the models used for the MIP method where the power model coefficients are given as deterministic values in an average sense, the power model coefficients are represented as rough variables which characterize the power model imprecision due to both power measurement and the simplifications made in order to cope with the complexity of the VLIW architecture. We have addressed the methodological issues of obtaining such a rough power model from the power modelling measurement data. This rough power model is then used to obtain the optimal schedule with minimal power variation.

The optimization problem is reformulated as a chance-constrained rough program instead of an MIP. The resulting optimal schedule is guaranteed to produce minimal power variation with a specified level of confidence, even considering the uncertainty due

to the power model imprecision characterized by the rough power model. In the process of searching for the optimum, ranking and ordering of the candidate schedules, which involves employing rough simulation [24] to compute their objective function values, are required. Rough simulation is a computationally expensive process. We propose a rule-based approach to rank the candidate schedules very efficiently. By only comparing the differences between two given schedules themselves, the difference between their objective function values can be obtained using a set of rules. This rule extraction process is done offline only once for the rough power model of a target VLIW processor. Using the rule-based approach for ranking the candidate schedules, we have respectively developed a problem-specific genetic algorithm (GA) and branch-and-bound algorithm to perform the search for optimal schedules. While the GA cannot guarantee optimality, the branch-and-bound algorithm do and thus generally has higher time complexity. The theory associated with these methods are mathematically justified.

Performance evaluation is conducted using the C6711 with MediaBench and Trimaran benchmarks. Our results show that the optimal schedules obtained by our new method can respectively achieve over 24% and 32% less power variations than the power-balanced schedules obtained by the mixed-integer program approach. The computational requirement of the proposed solutions using rule-based schedule ranking are also shown to be low enough for use in production compilers of embedded VLIW systems.

8.1 Recommendations for Further Research

There are several issues that have not been addressed in this thesis.

Issues related to cache misses The techniques proposed in this thesis are for VLIW instruction scheduling at the compile time for minimum power variation of the target processor. However, the power variation during the execution of the program would

be affected by instruction and data cache misses which are not considered in this paper. Cache misses cause extra delay and thus processor power variation would also be different. An extension of current work is to develop techniques that can be used at the compile time to estimate and handle cache misses.

Power-aware task scheduling The impact of software on power consumption increases from the instruction level to the operating system level; the higher the level, the greater the impact [57]. We may draw ideas from the power-aware scheduling techniques proposed in this thesis to apply to power-aware task scheduling at the operating system level for an embedded system with multiple devices, where task requests are often generated concurrently. Although scheduling is a standard feature in operating systems, power-aware task scheduling needs to be investigated for OS-based power management.

Rough programming theory and applications Real-life decisions are usually made with incomplete information. Rough programming deals with situations where some or all of the parameters of the optimization problem are described by rough variables rather than by deterministic quantities. As introduced in Section 5.3, the basic idea used in rough programming is to convert the rough programming problem into an equivalent deterministic problem by using measures such as the trust measure, pessimistic/optimistic value or expected value, of the rough variables. Then the optimal solution is obtained based on the converted rough program. However, we realize that since the optimization problems involve uncertainty, it appears that it is more reasonable to have a transition stage from absolute optimal to absolute non-optimal. A future direction could be to develop new rough programming models aimed at describing such a situation.

Besides, our work in this thesis provides a practical example of applying the rough programming approach to the scheduling problem. In particular, we propose a rule-based approach to much more efficiently rank, in terms of their pessimistic values, the

candidates in the rough environment. The theoretical basis of our rule-based approach is derived rigorously. Therefore, similar ideas may be drawn from our rule-based approach to ranking the candidates in the rough environment in terms of the other measures of rough variables, e.g., trust measure, optimistic value or expected value, since rough simulation is a very computationally expensive process and thus generally impractical in real applications.

Appendix A

Effects of the number of parallel instructions on Power Consumption for the C6711

The C6711 is an 8-issue VLIW digital signal processor. Table A.1, quoted from [43], gives the power consumption of the TMS320C6711 CPU (150MHz) and internal memory at 50% high / 50% low activity level and 75% high / 25% low activity level separately. Table A.2 describes the characteristics, i.e. CPU activity level, program and data memory access rate, associated with high activity level and low activity level separately. Note that the high activity level is characterized with a CPU activity level of eight parallel instructions; the low activity level is characterized with a CPU activity level of two parallel instructions.

We can compute the average CPU activity level associated with 50% high / 50% low

Table A.1: Power consumption statistics of TMS320C6711 CPU (150MHz) and internal memory.

Activity Level	CPU and Internal Memory (W)
50% High, 50% Low	0.73
75% High, 25% Low	0.87

Table A.2: Characteristics of activity levels of TMS320C6711.

Activity Level	CPU Activity Level	Program Memory Access Rate	Data Memory Access Rate
High	8 instructions	100%	100% CPU, 50% DMA
Low	2 instructions	25%	12.5% CPU

Table A.3: Power consumption statistics of TMS320C6711 CPU (150MHz) and internal memory in various average CPU activity level.

Activity Level	Average CPU Activity Level	CPU and Internal Memory (W)
50% High, 50% Low	5 instructions	0.73
75% High, 25% Low	6.5 instructions	0.87
100% High, 0% Low	8 instructions	1.01
0% High, 100% Low	2 instructions	0.45

activity level and 75% high / 25% low activity level respectively. They are $8 \times 50\% + 2 \times 50\% = 5$ and $8 \times 75\% + 2 \times 25\% = 6.5$ respectively, as shown in Table A.3. The power consumption consumed by the TMS320C6711 CPU (150MHz) and internal memory, at 50% high / 50% low activity level and 75% high / 25% low activity level respectively have been given in Table A.1. Thus the difference of the power consumption of the TMS320C6711 CPU (150MHz) and internal memory in the average CPU activity level of 6.5 and 5 parallel instructions respectively is 0.14W.

We can compute the power consumption of CPU and internal memory at 100% high / 0% low activity level and 0% high / 100% low activity level separately by linear interpolation. The results are given in Table A.3. Then the difference of the power consumption of the TMS320C6711 CPU (150MHz) and internal memory in the average CPU activity level of 8 and 2 parallel instructions respectively is 0.56W.

These data give us an idea of the large impact of the number of parallel instructions on the power consumption (or equivalently current) of CPU and internal memory.

Besides, the data in Table A.3 also show that VLIW instruction scheduling for performance and for total energy minimization do not conflict. For example, when the average CPU activity is increased by four times, from two parallel instructions to eight, the power dissipation of the CPU and internal memory only increase by 2.24 times.

Appendix B

Scheduling for performance and for total energy minimization on C6711

Our experiment is conducted with C6711, a VLIW digital signal processor which is introduced in Section 2.1.2. Figure 2.2 illustrates its internal organization. The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, and .D).

Our goal is to exam whether there are conflicts between VLIW instruction scheduling

```

LOOP :
      ADD  .L1  A1, A2, A3
      ||    ADD  .L2  B1, B2, B3
      ADD  .L1  A1, A2, A3
      ||    ADD  .L2  B1, B2, B3
      ...    ...
      ADD  .L1  A1, A2, A3
      ||    ADD  .L2  B1, B2, B3
      B    .S1    LOOP

```

Figure B.1: An C6711 assembly program which repeats two parallel “ADD”s.

```

LOOP :
    MPY  .M1  A1, A2, A3
||      MPY  .M2  B1, B2, B3
    MPY  .M1  A1, A2, A3
||      MPY  .M2  B1, B2, B3
    ...    ...
    MPY  .M1  A1, A2, A3
||      MPY  .M2  B1, B2, B3
    B     .S1    LOOP

```

Figure B.2: An C6711 assembly program which repeats two parallel “MPY”s.

```

LOOP :
    ADD  .L1  A1, A2, A3
||      ADD  .L2  B1, B2, B3
||      MPY  .M1  A1, A2, A3
||      MPY  .M2  B1, B2, B3
    ADD  .L1  A1, A2, A3
||      ADD  .L2  B1, B2, B3
||      MPY  .M1  A1, A2, A3
||      MPY  .M2  B1, B2, B3
    ...    ...
    ADD  .L1  A1, A2, A3
||      ADD  .L2  B1, B2, B3
||      MPY  .M1  A1, A2, A3
||      MPY  .M2  B1, B2, B3
    B     .S1    LOOP

```

Figure B.3: An C6711 assembly program which repeats four parallel instructions: two “ADD”s and two “MPY”s.

```

LOOP :
    ADD  .L1  A1, A2, A3
    ||   ADD  .L2  B1, B2, B3
    MPY  .M1  A1, A2, A3
    ||   MPY  .M2  B1, B2, B3
    ADD  .L1  A1, A2, A3
    ||   ADD  .L2  B1, B2, B3
    MPY  .M1  A1, A2, A3
    ||   MPY  .M2  B1, B2, B3
    ...  ...
    ADD  .L1  A1, A2, A3
    ||   ADD  .L2  B1, B2, B3
    MPY  .M1  A1, A2, A3
    ||   MPY  .M2  B1, B2, B3
    B    .S1    LOOP

```

Figure B.4: An C6711 assembly program which repeats two parallel “ADD”s and two parallel “MPY”s successively.

for performance and scheduling for total energy minimization.

We design our experiment as follows. First, we obtain the average current drawn through the core voltage CV_{DD} of TMS320C6711 is $402mA$ while running a program which repeats two “ADD” in parallel on L1 and L2. The program is given in Figure B.1. Appendix C contains an explanation of the assembly code structure of the C6711. Second, we obtain that average current drawn through the core voltage CV_{DD} of TMS320C6711 is $406mA$ while running a program which repeats two “MPY” in parallel on M1 and M2. The program is given in Figure B.2. Third, we obtain that average current drawn through the core voltage CV_{DD} of TMS320C6711 is $522mA$ while running a program which repeats two “ADD” and two “MPY” in parallel on L1, L2, M1 and M2. The program is given in Figure B.3.

Suppose that we shall schedule four instructions, two “ADD”s and two “MPY”s. The two programs given in Figure B.3 and B.4 are two possible schedules. Based on the results of the three experiments above, the average current drawn through the core voltage CV_{DD}

while executing the schedule in Figure B.4 would be $(402mA + 406mA)/2 = 404mA$. The four instructions, two “ADD”s and two “MPY”s, are scheduled to occupy two cycles. The total energy consumption can be computed as $CV_{DD} \times 404 \times 2$. In contrast, the average processor current while executing the schedule in Figure B.3 is $522mA$ as given in the third experiment above. But the four instructions are scheduled to occupy only one cycle, which is half reduced compared with that in the schedule in Figure B.4. The total energy consumption can be computed as $CV_{DD} \times 522$. Thus the total energy consumption is actually reduced.

Therefore VLIW instruction scheduling for performance and for total energy minimization do not conflict. VLIW instruction scheduling techniques for performance optimizations are adaptable to total energy minimization.

Appendix C

Assembly code structure of C6711

A line of C6711 assembly code has the following form which can include up to seven items [2].

label : *parallel bars* [*condition*] *instruction* *unit* *operands*
 ;*comments*

1. *Label*: A label identifies a line of code or a variable and represents a memory address that contains either an instruction or data.
2. *Parallel bars*: An instruction that executes in parallel with the previous instruction signifies this with parallel bars (||). This field is left blank for an instruction that does not execute in parallel with the previous instruction.
3. *Conditions*: All C6000 instructions are conditional. If no condition is specified, the instruction is always performed. If a condition is specified and that condition is true, the instruction executes. For example:

<i>With this condition...</i>	<i>The instruction executes if...</i>
[A1]	A1! = 0
[!A1]	A1 = 0

If a condition is specified and that condition is false, the instruction does not

execute. For example:

<i>With this condition...</i>	<i>The instruction does not execute if...</i>
[A1]	A1 = 0
[!A1]	A1! = 0

Five registers in the C62x/C67x are available for conditions: A1, A2, B0, B1, and B2. Six registers on the C64x are available for conditions: A0, A1, A2, B0, B1, and B2.

4. *Instruction*: Assembly code instructions are either directives or mnemonics. Assembler directives are commands for the assembler (asm6x) that control the assembly process or define the data structures (constants and variables) in the assembly language program. Processor mnemonics are the actual microprocessor instructions that execute at runtime and perform the operations in the program.
5. *Functional unit*: The C6000 CPU contains eight functional units, which are shown in Figure 2.2 and described above in this section.
6. *Operands*: The C6000 instructions use three types of operands to access data: 1) Register operands indicate a register that contains the data. 2) Constant operands specify the data within the assembly code. 3) Pointer operands contain addresses of data values. Only the load and store instructions require and use pointer operands to move data values between memory and a register. When an operand comes from the other register file, the unit includes an *X*, indicating that the instruction is using one of the cross paths.
7. *Comments*: As with all programming languages, comments provide code documentation.

Bibliography

- [1] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, “An instruction-level energy model for embedded VLIW architectures,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 9, pp. 998–1010, Sept. 2002.
- [2] Texas Instruments, “TMS320C6000 CPU and instruction set reference guide,” Texas Instruments Inc., Reference Guide SPRS088E, Oct. 2000.
- [3] J. T. Russell and M. Jacone, “Software power estimation and optimisation for high performance, 32-bit embedded processors,” in *Proc. Int. Conf. on Computer Design: VLSI in Computers & Processors*, Oct. 1998, pp. 328–333.
- [4] N. Julien, J. Laurent, E. Senn, and E. Martin, “Power consumption modeling and characterization of the TI C6201,” *IEEE Micro*, vol. 23, no. 5, pp. 40–49, Sep.-Oct. 2003.
- [5] H. Liao and A. Wolfe, “Available parallelism in video applications,” in *Proc. Int. Symp. on Microarchitecture*, 1997.
- [6] J. Fritts, W. Wolf, and B. Liu, “Understanding multimedia application characteristics for designing programmable media processors,” in *Proc. SPIE Photonics West, Media Processors*, San Jose, CA, Jan. 1999, pp. 2–13.
- [7] J. Fritts and W. Wolf, “Instruction fetch characteristics of media processing,” in *Proc. SPIE Photonics West, Media Processors*, San Jose, CA, Jan. 2002, pp. 72–83.

- [8] P. Faraboschi, J. A. Fisher, and C. Young, "Instruction scheduling for instruction level parallel processors," *Proceedings of the IEEE*, vol. 89, no. 11, pp. 1638–1659, Nov. 2001.
- [9] Y. S. Chang, S. K. Gupta, and M. A. Breuer, "Analysis of ground bounce in deep sub-micron circuits," in *VLSI Test Symposium, IEEE*, 1997, pp. 110–116.
- [10] M. Pant, P. Pant, D. Wills, and V. Tiwari., "An architectural solution for the inductive noise problem due to clock-gating," in *Proc. Int. Symp. on Low Power Electronics and Design*, 1999, pp. 255–257.
- [11] D. Brooks and M. Martonosi, "Dynamic thermal management for high-performance microprocessors," in *Proc. Int. Symp. on High-Performance Computer Architecture*, 2001, pp. 171–182.
- [12] A. Dhodapkar, C. H. Lim, and G. Cai, "Tem2p2est : A thermal enabled multi-model power/performance estimator," in *Proc. of Workshop on Power-Aware Computer Systems*, 2000, pp. 112–125.
- [13] M. Pedram and Q. Wu, "Battery-powered digital CMOS design," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 10, no. 5, pp. 601–607, Oct. 2002.
- [14] H. Yun and J. Kim, "Power-aware modulo scheduling for high-performance VLIW processors," in *Proc. Int. Symp. on Low Power Electronics and Design*, Huntington Beach, California, USA., Aug. 2001, pp. 40–45.
- [15] H. Yang, G. R. Gao, and C. Leung, "On achieving balanced power consumption in software pipelined loops," in *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, Grenoble, France, Oct. 2002, pp. 210–217.

- [16] V. Tiwari, S. Malik, and A. Wolfe, “Power analysis of embedded software a first step toward software power minimization,” *IEEE Trans. on Very Large Scale Integration Systems*, vol. 2, no. 4, pp. 437–445, Dec. 1994.
- [17] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, “Power exploration for embedded VLIW architectures,” in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, 2000, pp. 498–503.
- [18] A. Bona, M. Sami, D. Sciutos, C. Silvano, V. Zaccaria, and R. Zafalon, “Energy estimation and optimization of embedded VLIW processors based on instruction clustering,” in *Proc. 39th Design Automation Conference*, New Orleans, USA, June 2002 10–14,, pp. 886–891.
- [19] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, and R. Zafalon, “A framework for modeling and estimating the energy dissipation of VLIW-based embedded systems,” *Design Automation for Embedded Systems*, vol. 7, pp. 183–203, 2002.
- [20] V. Zaccaria, M. Sami, D. Sciuto, and C. Silvano, *Power estimation and optimization methodologies for VLIW-based embedded systems*. Boston: Kluwer, 2003.
- [21] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, “The design and use of SimplePower: a cycle-accurate energy estimation tool,” in *Proc. 37th Design Automation Conference*, Los Angeles, USA, June 2000 5–9,, pp. 340–345.
- [22] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A framework for architectural-level power analysis and optimizations,” in *Proc. Int. Symp. on Computer Arch.*, 2000, pp. 83–94.
- [23] V. Kathail, M. S. Schlansker, and B. R. Rau, “Compiling for EPIC architectures,” *Proceedings of the IEEE*, vol. 89, no. 11, pp. 1676–1693, Nov. 2001.

- [24] B. Liu, *Theory and practice of uncertain programming*. Heidelberg: Physica-Verlag, 2002.
- [25] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proc. Int. Symp. on Microarchitecture*, Dec. 1997, pp. 330–335.
- [26] L. N. Chakrapani, J. Gyllenhaal, W. mei W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah, "Trimaran: An infrastructure for research in instruction-level parallelism," in *Languages and Compilers for High Performance Computing: 17th International Workshop: Revised Selected Papers*, ser. Lecture Notes in Computer Science, R. Eigenmann, Z. Li, and S. P. Midkiff, Eds. Springer Verlag, 2005, vol. 3602, pp. 32–41.
- [27] M. S. Schlansker and B. R. Rau, "Epic: Explicitly parallel instruction computing," *Computer*, vol. 33, p. 37C45, 2000.
- [28] D. Landskov, S. Davidson, B. D. Shriver, and P. W. Mallett, "Local microcode compaction techniques," *ACM Computing Survey*, vol. 12, p. 261C294, Sept. 1980.
- [29] J. A. Fisher, "The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources," Ph.D. dissertation, Courant Mathematics and Computing Laboratory, New York Univ., New York, 1979.
- [30] J. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. on Computers*, vol. 30, p. 478C490, July 1981.
- [31] J. A. Fisher, D. Landskov, and B. D. Shriver, "Microcode compaction: Looking backward and looking forward," in *Proc. National Computer Conf.*, 1981, p. 95C102.

- [32] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proc. Annu. Microprogramming Workshop*, 1981, p. 183C198.
- [33] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The cydra 5 departmental supercomputer: Design philosophies, decisions and trade-offs," *Computer*, vol. 22, p. 12C35, 1989.
- [34] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, "Introducing the ia-64 architecture," *IEEE Micro*, vol. 20, p. 12C23, 2000.
- [35] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Computing Survey*, vol. 31, no. 3, pp. 264–323, Sept. 1999.
- [36] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*. New York: Wiley, 1976.
- [37] R. Allen and K. Kennedy, *Optimizing compiler for modern architectures: A dependence-based approach*. San Francisco: Morgan Kaufmann, 2002.
- [38] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Trans. on Computers*, vol. C-21, pp. 1405–1411, Dec. 1972.
- [39] Z. SHAO, B. XIAO, C. XUE, Q. ZHUGE, and E. H.-M. SHAE, "Loop scheduling with timing and switching-activity minimization for VLIW DSP," *ACM Trans. Design Automation of Electronic Systems*, vol. 11, no. 1, pp. 165–185, Jan. 2006.
- [40] C. Lee, J. K. Lee, T. T. Hwang, and S. C. Tsai, "Compiler optimization on VLIW instruction scheduling for low power," *ACM Trans. Design Automation of Electronic Systems*, vol. 8, no. 2, pp. 252–268, Apr. 2003.
- [41] C. Lee, J. K. Lee, and T. T. Hwang, "Compiler optimization on instruction scheduling for low power," in *Proc. Int. Symp. on System Synthesis*, 2000, pp. 55–60.

- [42] Texas Instruments, “TMS320C621x/C671x DSP two-level internal memory reference guide,” Texas Instruments Inc., Application Report SPRU609, Aug. 2002.
- [43] —, “TMS320C62x/C67x power consumption summary,” Texas Instruments Inc., Application Report SPRA486C, July 2002.
- [44] Z. Pawlak, *Rough Sets: theoretical aspects of reasoning about data*. Boston, MA: Kluwer, 1991.
- [45] Z. Pawlak, J. Grzymala-Busse, R. Slowinski, and W. Ziarko, “Rough sets,” *Communications of the ACM*, vol. 38, no. 6, pp. 88–95, June 1995.
- [46] Z. Pawlak, “Granularity of knowledge, indiscernibility and rough sets,” in *Proc. IEEE Int. Conf. on Fuzzy Systems*, 1998, pp. 106–110.
- [47] —, “Theorize with data using rough sets,” in *Proc. Annual Int. Computer Software and Applications Conf.*, 2002.
- [48] D. Kim and S. Bang, “A handwritten numeral character classification using tolerant rough set,” *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 22, no. 9, pp. 923–937, Sept. 2000.
- [49] L. Lazareck and S. Ramanna, “Classification of swallowing sound signals: A rough set approach,” in *Proc. Int. Conf. on Rough Sets and Current Trends in Computing*, Uppsala, Sweden, June 2004.
- [50] M. Ningler, G. Stockmanns, G. Schneider, O. Dressler, and E. F. Kochs, “Rough set-based classification of eeg-signals to detect intraoperative awareness: Comparison of fuzzy and crisp discretization of real value attributes,” in *Proc. Int. Conf. on Rough Sets and Current Trends in Computing*, Uppsala, Sweden, June 2004.

- [51] S. Ramanna, J. F. Peters, and T. Ahn, “Software quality knowledge discovery: A rough set approach,” in *Proc. Annual Int. Computer Software and Applications Conf.*, Oxford, England, Aug. 2002.
- [52] F. E. H. Tay and L. Shen, “Economic and financial prediction using rough sets model,” *European Journal of Operational Research*, vol. 141, pp. 643–661, 2002.
- [53] A. Øhrn, “Discernibility and rough sets in medicine: Tools and applications,” Ph.D. dissertation, Department of computer and information science, Morwegian University of science and technology, Trondheim, Norway, 1999.
- [54] H. S. Nguyen, “Discretization problem for rough sets methods,” in *Proc. Int. Conf. on Rough Sets and Current Trends in Computing*, Warsaw, Poland, June 1998, pp. 545–552.
- [55] J. Dougherty, R. Kohavi, and M. Sahami, “Supervised and unsupervised discretization of continuous features,” in *Proc. Int. Conf. on Machine Learning*. San Francisco, CA: Morgan Kaufmann, 1995.
- [56] D. E. Goldberg, *Genetic algorithms in search, optimization and machine learning*. Boston, MA: Addison-Wesley, 1989.
- [57] E. Macii, M. Pedram, and F. Somenzi, “High-level power modeling, estimation and optimization,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 11, pp. 1061–1079, Nov. 1998.

List of Publications

Journal

1. S. Xiao and E. M-K. Lai, "VLIW Instruction Scheduling for Minimal Power Variation," submitted to *ACM Trans. on Architecture and Code Optimization*, 2005.
2. S. Xiao and E. M-K. Lai, "A Rough Programming Approach to Power-Balanced Instruction Scheduling for VLIW Digital Signal Processors," submitted to *IEEE Trans. on Signal Processing*, 2005.

Conferences

1. S. Xiao, E. M-K. Lai and A. B. Premkumar, "Rule-based power-balanced VLIW instruction scheduling with uncertainty," *Proceedings of 10th Asia-Pacific Computer Systems Architecture Conference*, Singapore, Oct. 2005.
2. S. Xiao, E. M-K. Lai and V. A. Prasad, "A rough programming solution to VLIW instruction scheduling," *Proceedings of 8th International Symposium on Signal Processing and Its Applications*, Sydney, Australia, Aug. 2005.
3. S. Xiao and E. M-K. Lai, "Power-balanced VLIW instruction scheduling using rough programming," *Proceedings of 4th International Conference on Information and Management Science*, Kunming, China, Jul. 2005.

4. S. Xiao and E. M-K. Lai, "A rough set approach to instruction-level power analysis of embedded VLIW processors," *Proceedings of 4th International Conference on Information and Management Science*, Kunming, China, Jul. 2005.
5. S. Xiao and E. M-K. Lai, "A rough programming approach to power-aware instruction scheduling for digital signal processors," *Proceedings 30th International Conference on Acoustics, Speech and Signal Processing*, Philadelphia, U.S.A., Mar. 2005.
6. S. Xiao and E. M-K. Lai, "Instruction scheduling of VLIW architectures for balanced power consumption," *Proceedings of 10th Asia and South Pacific Design Automation Conference*, Shanghai, China, Jan. 2005.
7. S. Xiao and E. M-K. Lai, "Power-aware VLIW instruction scheduling," *Proceedings of 3th International Conference on Information*, Tokyo, Japan, Nov. 2004.
8. S. Xiao and E. M-K. Lai, "A branch and bound algorithm for fast power-aware scheduling of VLIW instructions," *Proceedings of Workshop on Compilers and Tools for Constrained Embedded Systems* (in conjunction with 7th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems), Washington D.C., U.S.A., Sep. 2004.