10-24-2003

Force-Directed Instruction Scheduling for Low Power

Prashant Dongale

University of South Florida

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the American Studies Commons

Scholar Commons Citation

https://scholarcommons.usf.edu/etd/1357

This Thesis is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
Force-Directed Instruction Scheduling for Low Power

by

Prashant Dongale

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science Department of Computer Science and Engineering College of Engineering University of South Florida

Major Professor: N.Ranganathan, Ph.D. Murali Varanasi, Ph.D. Dewey Rundus, Ph.D.

Date of Approval: October 24, 2003

Keywords: optimization, estimation

© Copyright 2003, Prashant Dongale
DEDICATION

This work is dedicated to Aai, Papa, Archana and Prasad.
ACKNOWLEDGEMENTS

I would like to thank Dr. N. Ranganathan for giving me this opportunity to work with him. I owe him a lot for having given directions to my thoughts and unlimited encouragement. Without his insights and patient guidance this work would not have been possible. He has been a steadfast support throughout my Master’s program. I also wish to thank Dr. Murali Varanasi & Dr. Dewey Rundus for the time they took to review this thesis and their helpful comments.

I am very grateful to my loving Father, Mother, and my siblings: Archana & Prasad, without whose moral support this work would never have reached the completion. I am indebted to them for being a perpetual source of inspiration and motivation for me. I cannot forget the help and support of all my friends and colleagues who have helped in every step of the way. Special thanks to my roommates: Ayush and Sanky for their support, all the way through.
TABLE OF CONTENTS

LIST OF TABLES iii

LIST OF FIGURES iv

ABSTRACT v

CHAPTER 1 INTRODUCTION 1
  1.1 Sources of Software Power Consumption 3
  1.2 Power Optimization at Different Levels 3
  1.3 Thesis Contribution 4
  1.4 Thesis Organization 7

CHAPTER 2 RELATED WORK 8
  2.1 Software Power Estimation 8
    2.1.1 Gate-Level Power Estimation 8
    2.1.2 Architecture-Level Power Estimation 9
    2.1.3 Bus Switching Activity 10
    2.1.4 Instruction-Level Power Analysis 10
  2.2 Software Power Optimization 12
    2.2.1 Instruction-Level Optimization 12
    2.2.2 Compiler-Level Optimization 12
      2.2.2.1 Instruction Scheduling 13
      2.2.2.2 Energy Cost Based Code Generation 13
      2.2.2.3 Reducing Memory Accesses 14
      2.2.2.4 Source Code Transformations 14
      2.2.2.5 Instruction Packing, Dual Memory Loads and Operand Swapping 14
    2.2.3 System-Level Power Optimization 15
  2.3 Instruction Scheduling for Low Power 15
    2.3.1 Gray Code Addressing and Cold Scheduling 15
    2.3.2 Instruction Scheduling as a TSP 17
    2.3.3 Instruction Scheduling to Reduce Off-Chip Power 17
    2.3.4 Minimize Peak Power Dissipation 18
    2.3.5 Energy and Performance Instruction Scheduling 18
  2.4 Classical Force-Directed Scheduling 19
  2.5 Our Work 22
  2.6 Summary 23
LIST OF TABLES

Table 2.1 Bit Switching in Binary and Gray Code Representation 16
Table 3.1 Probabilities of Instruction 2 for Assignment (2,2) 32
Table 3.2 Possible Assignments for Instruction 2 32
Table 3.3 Probabilities of Instruction 3 for Assignment (2,2) 33
Table 3.4 Possible Assignments for Instruction 3 33
Table 4.1 Instruction Set Architecture 37
Table 4.2 Benchmark Set 38
Table 4.3 Power Savings Results for Benchmarks 40
Table 4.4 Execution Time of Benchmarks 40
Table 4.5 Power Savings for Variable Sized Input Vectors 41
Table 4.6 Power Consumption Breakdown for Binary Search Benchmark 41
Table 4.7 Power Savings Breakdown for Quick Sort 41
Table 4.8 Power Savings Breakdown for Matrix Multiplication 42
Table 4.9 Power Savings Breakdown for Heap Sort 42
Table 4.10 Power Savings Comparison 42
Table 4.11 Savings in Running Time 43
LIST OF FIGURES

Figure 1.1  Trends in CPU Power Consumption 1
Figure 1.2  The Cost of Removing Heat from a Microprocessor 2
Figure 1.3  Flow Diagram for Power Optimization Methodology 6
Figure 2.1  Taxonomy of Software-Power Related Works 9
Figure 2.2  Steps Involved in Instruction-Level Power Analysis 11
Figure 2.3  Unscheduled Data Flow Graph 20
Figure 2.4  ASAP Schedule for DFG in Figure 2.3 21
Figure 2.5  ALAP Schedule for DFG in Figure 2.3 21
Figure 3.1  PDT Generation Example 26
Figure 3.2  Basic Block Partitioning Algorithm 27
Figure 3.3  Basic Block Generation Example 28
Figure 3.4  DDG Generation Example 29
Figure 3.5  FD-ISLP Algorithm 30
Figure 3.6  An Example of a DDG 31
Figure 3.7  Partial Schedule for the DDG in Figure 3.6 31
Figure 4.1  Power Optimization and Validation Framework 36
FORCE-DIRECTED INSTRUCTION SCHEDULING FOR LOW POWER

Prashant Dongale

ABSTRACT

The increasing need for low-power computing devices has led to the efforts to optimize power in all the components of a system. It is possible to achieve significant power optimization at the software level through instruction reordering during the compilation phase. In this thesis, we have designed and implemented a novel instruction scheduling technique, called FD-ISLP, aimed at reducing the software power consumption. In the proposed approach for instruction scheduling, we modify the force-directed scheduling technique used in high-level synthesis of VLSI circuits to derive a latency-constrained algorithm that reorders the instructions in a basic block of assembly code in application software to reduce power consumption due to its execution. The scheduling algorithm takes the data dependency graph (DDG) for a given basic block and a power dissipation table (PDT), which is generated by characterizing the instruction set architecture. We model power, commonly referred to as software power in literature, as a force to be minimized by relating the inter-instruction power cost as the spring constant, \( k \), and the change in instruction probability as the displacement, \( x \), in the force equation \( f = k \cdot x \). The salient feature of our algorithm is that it accounts for the global effect of any tentative scheduling decision, which avoids a solution being trapped in a local minima. The power estimates are obtained through using a tool set, called SimplePower. Experimental results indicate that our technique accounts for an average of 12.68% savings in power consumption over the original source code for the selected benchmark programs.
CHAPTER 1
INTRODUCTION

The traditional concerns for system design have been performance, reliability and cost. However, with the advent and tremendous growth in the need for personal computing devices and wireless communication systems, power consumption is becoming a major concern in the design of modern systems. Low power has become a key feature for most of the portable devices. Therefore, reducing the power consumption of such devices elongates the battery lifetime and hence the device up-time. The power consumption of Intel processors over the period of 15 years has been shown in Figure 1.1 (source:[10]). As can be seen from the trend that the maximum power consumption tends to increase by a factor more than 2X every four years [10].

![Power Trends](image)

**Figure 1.1. Trends in CPU Power Consumption**

Modern processors running at high clock rates consume large amount of power and the cost associated with cooling and packaging such devices is huge. The core power consumption needs to be dissipated through packaging, which requires increasingly expensive cooling and packaging tech-
nologies. The cost of removing heat from processors having different levels of power consumption is shown in Figure 1.2 (source: [10]). It can be observed from the figure that as power consumption increases, the cost associated with cooling solution increases in a non-linear fashion. This stresses on the importance of limiting the power consumption to a threshold defined by the cost structure of the platform. Therefore, it is important to reduce the power consumption as the resulting heat limits feasible packaging and performance of VLSI circuits and systems.

![Figure 1.2. The Cost of Removing Heat from a Microprocessor](image)

There is also the issue of reliability of the systems. High power systems often run hot, which aggravates silicon failure mechanisms. Every $10^\circ$C increase in temperature roughly doubles the failure rate of the components [35]. In this regard, peak power becomes an important issue, which needs to be kept under control. Also, there are environmental issues that prompt us for low power systems.

All these issues discussed earlier emphasize on the design of the systems with reduced power consumption, although the motivations for power reduction differ from application to application. In the next section we will look at the sources of power consumption that are most influenced by software.
1.1 Sources of Software Power Consumption

Software significantly impacts the following parts of the CPU, which are major contributors towards the CPU power consumption: Memory system, System busses, Datapaths.

Memory system shares a major fraction of CPU power consumption in portable computers and especially for memory intensive DSP applications such as video processing. Accessing the memory locations involves switching on highly capacitive data and address lines to the memory, decode logic, word and data lines within the memory. Also, the memory access patterns of a program largely determine the cache performance of a system. Low cache performance because of high number of cache misses lead to expensive memory accesses. Compact machine codes can also help to reduce the memory accesses by reducing the probability of cache misses [13].

System busses such as address, instruction and data busses have high load capacitances. Sequence of instruction op-codes determines the switching activity on an instruction bus. Similarly, switching activity on data and address busses is influenced by the sequence of data and instruction accesses, respectively. The data related switching is hard to predict at compile time because most input data to a program is provided only at the execution time.

Datapaths to arithmetic logic units and floating point units constitute a substantial fraction of the logic power dissipation in a CPU. They are demanding of power, especially if all operational units or pipeline stages are activated at all times. Selection of instructions, scheduling of operations are some examples of software design decisions that can affect datapath power.

1.2 Power Optimization at Different Levels

Power optimizations can be achieved at different levels such as circuit, logic, architectural, system and software levels. At the circuit level there are techniques such as reordering of gates and transistor sizing. In [30] authors have given method to optimize the power of logic-gates based on transistor reordering. At logic level, different techniques are used for combinational and sequential circuits. Combinational circuits use path balancing and factorization techniques. In order to reduce the spurious transitions, the delays of all paths that converge at each gate should be roughly equal. Path balancing involves adding unit-delay buffers to inputs of gates to make the delays equal. Factorization involves finding common subexpressions and reusing them, which can reduce the tran-
sistor count. Sequential circuits use techniques such as encoding, retiming and precomputation to optimize power. Retiming is a well-known optimization technique that repositions the flip-flops in a sequential circuit so as to minimizing the required clock period. A retiming method that targets the power optimization has been presented in [25]. In precomputation, the output logic values of a circuit are selectively precomputed one clock cycle before they are required, and these values are used to turn off the load signal to the registers whose output does not change in next cycle. This helps to reduce the power dissipation of the circuit.

During behavioral synthesis, high-level specifications are modified by specific transformations, which lead to power reduction. The most important transformations are those which reduce the number of control steps, amount of resources required and switched capacitance. Power optimization can also be achieved at the software level. Software impacts the power consumption of a system, on which it runs, to a great extent. The most fundamental level at which software power can be controlled is the instruction level, because different software instructions cause switching at different hardware modules and hence consume variable amounts of power. For power optimization at instruction level, there are several techniques such as instruction scheduling, reducing memory access, energy cost driven code generation, and instruction packing. The order of instructions can have an impact on power since it determines internal switching in the CPU. Therefore selecting a proper schedule of instructions can reduce power. Memory operands are more expensive than register operands in terms of power consumption. Therefore optimal register allocation that reduces the memory operands, can significantly reduce power. If power costs of individual instructions are known, an appropriate choice of instructions during code generation can lead to reduction in power cost. Software controlled voltage and frequency scaling are some other examples of techniques used at the software level for dynamic power optimization.

1.3 Thesis Contribution

As we know from discussion in the previous section that the nature of the instructions and their reordering in the given software influence the power consumption of a system, it is increasingly important to analyze and optimize the system power consumption from a software point of view. Just as logic gates are the fundamental units in digital hardware circuits, instructions can be the
fundamental units in software. Therefore, an accurate modeling and analysis at instruction level is essential for quantifying power at higher levels of software abstraction. We are using the instruction model presented in [49] for our work.

In this thesis, we describe a new instruction scheduling algorithm for software power optimization. Our approach for instruction scheduling is based on the classical force-directed scheduling (FDS) technique presented by Paulin and Knight [29]. FDS has been effectively applied in behavioral and high-level synthesis of VLSI circuits in the field of VLSI design automation. A modification of the FDS algorithm for power optimization during behavioral synthesis, referred to as FDS-LP algorithm, is described in [12] and [11]. Thus, applying the force directed scheduling technique for instruction reordering at the compiler level is being addressed for the first time in this work.

The classical FDS is a latency constrained scheduling technique that attempts to minimize the resource requirements, by maximizing the concurrency of operations. The FDS algorithm presented by Paulin and Knight in [29] optimizes the resource requirements during synthesis, but does not address the issue of power consumption. We have modified the FDS to target power optimization during instruction scheduling and it can be applied at the compiler level. The motivation of using the FDS in instruction scheduling for low power lies in the ability to model power consumption as a force for optimization. We refer to our proposed force directed instruction scheduling algorithm for low power as FD-ISLP. The force \( f \) in the force equation \( f = kx \), is expressed as the product of spring constant \( k \) and displacement \( x \). We model the inter-instruction power cost as \( k \) and the change in instruction probability as \( x \), while modeling power as \( f \), which can be formulated in the following Equation 1.1.

\[
\begin{align*}
  f &= k \times x \\
  \text{power} &= \text{switching activity} \times \text{change in mobility}
\end{align*}
\]

The flow diagram for power optimization during instruction scheduling and its context with the compilation process is shown in Figure 1.3. The input source code is compiled and the derived assembly code is further divided into basic blocks. The given instruction set architecture (ISA) is power characterized to build a power dissipation table (PDT). Such a table for an \( n \) instruction ISA is a \( n \times n \) matrix, where each entry \((i, j)\) of the table represents the power dissipation when
instruction $i$ is followed by instruction $j$. A data dependency graph (DDG) is constructed for each basic block. The Low-Power Force Directed Instruction Scheduling (FD-ISLP) algorithm tries to reschedule each basic block of the source program for low power. The FD-ISLP takes as input a DDG for the current basic block and the PDT. During each iteration of the algorithm, it finds all the instructions that can be scheduled in that time-step, using the DDG. It tentatively assigns an instruction to a time-step and computes the force (i.e. power cost) associated with that assignment. Finally, the instruction having the least force value is chosen to be scheduled in the current time-step. These steps are followed till all the instructions in the given basic block are scheduled.

![Flow Diagram for Power Optimization Methodology](image)

**Figure 1.3. Flow Diagram for Power Optimization Methodology**
The SimpleScalar ISA was considered and the power simulations were performed using SimplePower simulator for 0.35 micron technology. Five benchmark programs were used to verify the effectiveness of the scheduling algorithm.

1.4 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 will discuss the related work, which has been previously done in software power optimization with a special emphasis on instruction level techniques and also the classical FDS algorithm, which forms the basis of our work. Chapter 3 will discuss the proposed instruction scheduling algorithm for low power. Chapter 4 will report simulation environment, experimental results and their analysis. Chapter 5 will present conclusions and the future directions that can be explored augmenting out work.
CHAPTER 2

RELATED WORK

Software directs much of the activity of the hardware in microprocessor, micro-controller and digital signal processor based systems. Consequently, software can have a substantial effect on the power consumption of a system. In light of this, there is a clear need to consider the power consumption of the system from a software point of view. Also, the ability to evaluate software in terms of power consumption can help to verify that a particular design meets its power constraints.

The first step towards software power optimization is software power estimation. A number of techniques have been proposed for software power estimation and optimization, some of which are briefly discussed in this chapter. The taxonomy of the software power related works is presented in Figure 2.1. The related works are classified based on the different levels of the system at which they are applied (estimation: gate, architecture, instruction level and for optimization: instruction, compiler and system level).

2.1 Software Power Estimation

Software power is estimated using different approaches: gate-level estimation, architecture-level estimation, switching activity on buses and instruction-level analysis [31].

2.1.1 Gate-Level Power Estimation

In [26], [7], the authors have presented a gate-level power estimation model of a processor running a program. It is an accurate method but requires detailed gate-level description of the processor. Usually, such a detailed description is not available to the software developers and even if it is available, this method is too slow because of its high computational complexity.
2.1.2 Architecture-Level Power Estimation

At the architecture level, power estimation is less accurate but much faster than at gate-level. This approach requires a model of the processor at component level (such as ALU, register file, etc.) and the power dissipation estimates for those components. It also needs the knowledge about which components are activated by the individual instructions being executed. Finally, the execution of a program is simulated to identify the components that are active in each execution cycle and power dissipation of each component is added up to get the power estimate. Such an approach is implemented in [33], [22]. Mizuno et al. [24] have proposed an approach to the power estimation of an embedded hardware/software co-design system at the architectural level. The distinctive feature
of this approach is constructing a precise energy model at the architectural level for each component of embedded system, by taking into account data transfer as well as functional performance.

2.1.3 Bus Switching Activity

It is also important to estimate the power based on the bus switching activity of a processor. Modeling bus switching activity requires the knowledge of the bus architecture of a processor, binary op-codes of the instructions, the input data environment, and the manner in which code and data are mapped into the address space. For a given set of input instructions, switching statistics can be found by simulating an execution and monitoring the number of bit switches on each of the buses. Su et. al [36] have presented a cold scheduling algorithm for instruction scheduling that implements bus switching activity. In this work, only instruction and address related switching are considered due to the unpredictability of the data values.

2.1.4 Instruction-Level Power Analysis

Instruction-Level Power Analysis (ILPA) [39] presents a methodology for developing and validating an instruction level power model for a given processor. This method measures the current drawn by the processor as it repeatedly executes certain instructions.

In this model, each instruction in the instruction set architecture is associated with a fixed energy cost called the base energy cost. The base power cost is calculated as product of $V_{CC}$ and the average current drawn by the processor while running a loop with several instances of the same instruction. The base power cost times the number of non-overlapped cycles required to execute the instruction is its base energy cost. The variation in base energy due to different address values and operands is also computed. During the program execution there are certain inter-instruction effects, which are also quantified. The inter-instruction effects take into account effect of circuit state, effect of resource constraints e.g. pipeline stalls and write buffer stalls [16] and effect of cache misses. The circuit state overhead for a pair of instructions is defined as the difference between the actual cost of the pair and the average of the base cost of the individual instructions [41].

The overall energy cost of a program is computed as the sum of base energy costs of all participating instructions and their inter-instruction effects. The total energy cost, $E_p$ of a program can be
given by the following equation [41]:

\[
E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k
\]  

(2.1)

where for each instruction \(i\), \(B_i\) is the base cost and \(N_i\) is the number of times it will get executed, and for every pair of consecutive instructions \((i,j)\), \(O_{i,j}\) is the circuit state overhead, \(N_{i,j}\) is the number of times the pair will be executed. \(E_k\) is the energy of the other inter-instruction effects, that would occur during program execution. The overall flow of ILPA methodology is shown in Figure 2.2. This technique has been verified with two commercial microprocessors: the Intel 486DX2 [40] and the Fujitsu SPARCLite934 [37].

Mehta et al. [22], [21] have proposed a method that is based on a simulation of microprocessor and the effect of the instruction set in the microprocessor model. This work suggests that lower level simulation can provide an estimate of the current drawn to calculate the power consumption of each instruction. Chakrabarti et al. [5] have built a model of each basic module for the HC11

Figure 2.2. Steps Involved in Instruction-Level Power Analysis
micro-controller. The model has been implemented using hardware description languages. Also, black box models [8] or other kind of models, where it is possible to make a current or power measurement, can be used. Once, the modules activated by each instruction are determined, the power consumption can be computed as the sum of power consumption of each active module.

Sarta et al. [44] have presented an instruction level power analysis technique, that tries to relate the power dissipation to the executed instructions and their operand values. This model is shown to be very accurate for the processors, in which operand distribution strongly influences the power consumption.

2.2 Software Power Optimization

Software power can be controlled at the following different levels: instruction, compiler and the system level.

2.2.1 Instruction-Level Optimization

Instruction set optimization is an important issue in the hardware/software co-design domain. Efforts at instruction set optimization are presented by Sato et al. [32], Binh et al. [3] and Huang et al. [15]. The work of Sato and Binh are both related to the PEAS-I system (Practical Environment for ASIP development version-I). PEAS-I defines an extended set of instructions by starting with a core instruction set. For each possible instruction, alternative implementations are defined. Optimal selection of implementation is based on power and area constraints. Binh [3] extended the instruction selection formulation to account for pipeline hazards.

2.2.2 Compiler-Level Optimization

Compiler level power optimization techniques can further be classified into following categories: instruction scheduling for low power, selection of least expensive instructions during code generation, reducing the frequency or cost of memory accesses, source code transformations and exploiting the power reduction features of the hardware.
2.2.2.1 Instruction Scheduling

Scheduling algorithms can be classified as transformational or constructive. A transformational algorithm starts with a default schedule and applies a set of transformations that are allowed by the dependency constraints, to obtain a new schedule. On the other hand, a constructive algorithm builds a schedule by adding new operations to the existing schedule in each iteration. Instruction scheduling for low power aims at reducing the circuit state overhead. From the previous section, it is clear that circuit state overhead is the energy dissipated due to switching from execution of one type of instruction to another. Given the circuit state costs for all possible instruction pairs, it is possible to compute the power consumption of a sequence of instructions. Instructions can be reordered [38] to have less amount of circuit state overhead without considerable penalty in runtime. The experimental results depict that the power savings in DSPs due to instruction scheduling are much more significant compared to general purpose architectures. This is attributed to the fact that circuit state overhead constitutes a substantial part of the total power dissipation for DSPs.

Su et al. [36] have proposed a cold scheduling algorithm to reduce the switching activity along the control path. In [49], the authors have presented another scheduling algorithm by formulating the instruction scheduling problem as a traveling salesman problem (TSP). The algorithm uses minimum spanning tree and simulated annealing techniques for optimization. Instruction scheduling to reduce the off-chip power was accomplished by Tomiyama et al. [43]. This work tries to minimize the power consumption by reducing the bit-switching on the off-chip buses. A scheduling algorithm to reduce the peak power dissipation is presented in [42]. Energy and performance constraints for scheduling were combined by Parikh et al. [28].

2.2.2.2 Energy Cost Based Code Generation

The traditional approach selects instructions based on their code size or execution time. The low-power approach proposes the selection of instructions based on their energy cost during the code generation. It is observed that energy based and the running time based code generators produce almost similar code for 486DX2. However, this observation may not hold true for many application specific processors such as DSP cores. In such cases aggressive power management can be obtained at some overhead in program running time.


2.2.2.3 Reducing Memory Accesses

It has been observed that the power consumption of the instructions involving memory access is much higher than the instructions accessing only registers. For cache misses the cost is much higher due to multiple cache fill cycles and energy dissipation in the external memory system. Consequently, a large amount of energy can be saved by reducing the memory accesses, which can be achieved through optimum register allocation during the compilation phase.

2.2.2.4 Source Code Transformations

Simunic et al. [34] have presented an optimization methodology at three levels of abstraction: algorithmic, data and instruction-level. Algorithmic optimizations include implementation of different algorithms having same functionality and the comparison of their algorithmic efficiency (e.g. number of operations). The data and instruction-level optimization have also shown significant amount of power savings. The problem of generating compact and efficient code for embedded DSP processors has been addressed in [17]. The improvements in execution efficiency and code size are achieved by exploiting the the parallelism in the instruction set architecture. These improvements have also led to reduction in power consumption. In [45], a loop unrolling technique to reduce power consumption has been successfully applied to the DSP processors. In this work, the objective was to reduce the total number of comparisons, which resulted in low use of the ALU and power savings. Hong et al. [14] have developed a divide-and-conquer compilation method to minimize the number of operations for general DSP computations.

2.2.2.5 Instruction Packing, Dual Memory Loads and Operand Swapping

The DSPs have a special architectural feature that supports packing certain pair of instructions into a single instruction. The packed instruction gives the same functionality as the unpacked instruction pair, however, it consumes considerably lower power than the individual instructions executed sequentially. The use of dual data memories can also reduce the power consumption. A special dual load instruction can load two operands from two different memory banks in one cycle. Thus, the memory allocation techniques that make efficient use of the dual memory banks can save a considerable amount of memory access power. It is observed that for the Fujitsu DSP, appropriate
swapping of two multiplication operands could lead to up to 30% reduction in the multiplication power cost[20]. Under the software controlled power management technique, certain unneeded parts of the CPU can be powered down. This is done by setting appropriate bits inside system control register. The observations [41] show that this technique can save significant power for the specific processors, which support the necessary feature of powering down individual modules.

2.2.3 System-Level Power Optimization

At the system level, several memory optimization techniques have been proposed [2], [19]. In [18], [9], the authors have presented hardware-software partitioning techniques for constructing power-optimized processors. Voltage scaling is another technique proposed for power optimization. It is achieved through use of dynamically variable voltage supply [27],[6].

2.3 Instruction Scheduling for Low Power

In this section, we will look at existing instruction scheduling techniques for low power, in detail.

2.3.1 Gray Code Addressing and Cold Scheduling

Su et al. [36] have proposed two techniques, gray code addressing and cold scheduling for low power. Gray code has a one-bit difference in representation for consecutive numbers. Due to temporal and spatial localities, most of the program instructions access the consecutively addressed memory locations. Therefore, the use of gray code addressing can significantly reduce the number of bit switches along the address buses, saving considerable amount of power. Table 2.1 shows the number of bit switches for a stream of 16 numbers in binary addressing and gray code addressing schemes. Let $B = < b_{n-1}, b_{n-2}, ..., b_1, b_0 >$ be the binary representation and $G = < g_{n-1}, g_{n-2}, ..., g_1, g_0 >$ be the gray code representation of the same memory address. Then the following formula gives the conversion rule between binary and gray code representations.

\[
g_n = b_n \\
g_i = b_{i+1} \oplus b_i \quad (i = n-1,n-2,...,0)
\]
Table 2.1. Bit Switching in Binary and Gray Code Representation

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Gray Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
<td>00000</td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>00001</td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
<td>00011</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
<td>00010</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
<td>00110</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
<td>00111</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
<td>00101</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
<td>00100</td>
</tr>
<tr>
<td>8</td>
<td>01000</td>
<td>01100</td>
</tr>
<tr>
<td>9</td>
<td>01001</td>
<td>01101</td>
</tr>
<tr>
<td>10</td>
<td>01010</td>
<td>01111</td>
</tr>
<tr>
<td>11</td>
<td>01011</td>
<td>01110</td>
</tr>
<tr>
<td>12</td>
<td>01100</td>
<td>01010</td>
</tr>
<tr>
<td>13</td>
<td>01101</td>
<td>01011</td>
</tr>
<tr>
<td>14</td>
<td>01110</td>
<td>01001</td>
</tr>
<tr>
<td>15</td>
<td>01111</td>
<td>01000</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td>11000</td>
</tr>
</tbody>
</table>

| Switched bits | 31 | 16 |

The other technique called *cold scheduling* is a software approach based on a traditional list scheduling algorithm modified to reduce the switching activity along the control path. Traditional instruction scheduling algorithms aim at improving performance. They schedule the instructions to reduce the pipeline stalls, avoid pipeline hazards or improve resource utilization [36]. These algorithms need to be modified to attain the new objective of low power consumption.

The basic steps followed in the list scheduling algorithm are as follows.

1. Divide the code into basic blocks. A basic block is a piece of code which has only one entry point and one exit point.

2. Form a Control Dependency Graph (CDG) and/or a Data Dependency Graph (DDG) for each Basic Block.

3. Schedule instructions in each Basic Block satisfying the resource constraints and the dependency constraints.

Cold scheduling is an implementation of the list scheduling algorithm with a heuristic to reduce the switching activity. Let $B = I_1, I_2, ..., I_n$ be the instruction sequence in a basic block $B$. $S(I_i, I_j)$ de-
notes the number of bit switches when instruction $I_i$ is immediately followed by $I_j$. Total switching activity in a basic block is defined as:

$$BS = \sum S(I_i, I_{i+1}) \quad i = 0, 1, ..., n - 1$$ (2.3)

Then the goal of the cold scheduling algorithm is to minimize the block switching activity, $BS$. For a program consisting of $k$ basic blocks $B_1, B_2, ..., B_k$ the cost function of the cold scheduler is given by equation 2.4:

$$\Theta = \frac{1}{k \times (w_1 \times BS_1 + w_2 \times BS_2 + ... + w_k \times BS_k)}$$ (2.4)

where $w_1, w_2, ..., w_k$ are the weights based on the execution frequency of each basic block and $BS_1, BS_2, ..., BS_k$ are the total number of bit switches of each of the basic blocks.

### 2.3.2 Instruction Scheduling as a TSP

Choi et al. have [49] have formulated the instruction reordering problem as precedence constrained Hamiltonian path problem for directed acyclic graphs (DAG) and traveling salesman problem (TSP), both of which are NP-Hard. For power optimization, minimum spanning tree (MST) and simulated annealing (SA) mechanisms are used. The scheduling technique uses a power dissipation table (PDT), which is generated by power simulations. A PDT for an instruction set with $n$ instructions is a $(n \times n)$ matrix, where each entry $(i,j)$ is the power consumption due to execution of instruction $I_i$ followed by instruction $I_j$. The scheduling algorithm uses a control flow and data dependency graph (CDG) for each basic block and the PDT. The problem of instruction reordering for low power is transformed to finding the tour of lowest cost (TSP). Initial solution is found as a MST using Prim’s algorithm and is further refined by local search heuristics such as 2-optimal and 3-optimal by using SA.

### 2.3.3 Instruction Scheduling to Reduce Off-Chip Power

Tomiyama et al.[43] have proposed an instruction scheduling technique to reduce power consumed for off-chip driving. The power dissipated by the off-chip drivers is proportional to the
number of bit switches along the wires of the off-chip buses. The scheduling algorithm schedules the instructions in each basic block such that the binary representations of two consecutive instructions in a memory block are less different. This reduces the number of transitions on the data bus and hence the power consumption.

2.3.4 Minimize Peak Power Dissipation

In [42], an instruction scheduling approach is proposed which focuses on reducing the peak power dissipation as opposed to power consumption. The peak power dissipation can be reduced by reducing the occurrences of current spikes during program execution. This approach tries to reduce such current spikes by limiting the amount of energy that can be dissipated in a given cycle. This scheduling algorithm is based on list scheduling. Once an instruction is ready to be scheduled and assigned to a functional unit (FU), the scheduler queries the power dissipation value. The scheduler then adds the value to the power dissipation of the current cycle, if the total exceeds the predefined threshold then the scheduler quits scheduling for the current cycle and begins next cycle with the instruction that caused violation in the previous cycle.

2.3.5 Energy and Performance Instruction Scheduling

Parikh et al. [28] have proposed performance-oriented, energy-oriented and combined approaches for instruction scheduling. The performance-oriented approach uses the traditional list scheduling algorithm with delay as the cost parameter. The energy-oriented approach also uses the list scheduling, but uses circuit-state effect (inter-instruction effect [39]) as the cost parameter. Each node of the DAG to be scheduled has a base cost and the edges denote the circuit-state effect. At each step, the scheduler picks up the next node with least circuit-state effect. In the combined approach, the scheduling decision is based on one parameter and the other parameter is considered only in the case of a tie. e.g. in Energy-with-Performance approach, the decisions are based on circuit-state effect and the delay parameter is used only when there is a tie with the circuit-state values.
2.4 Classical Force-Directed Scheduling

The Force-Directed Scheduling (FDS) algorithm, widely used in high-level synthesis, is an example of a constructive approach for scheduling. The constructive algorithm builds a schedule by adding new operations to the existing schedule in each iteration.

Our approach for instruction scheduling is based on the work by Paulin and Knight [29]. In the following section, we will look into the details of the FDS. FDS is a latency-constrained algorithm. The intent of the FDS, which aims to reduce the resource requirements by balancing the concurrency of the operations assigned to them without violating the latency constraints. The algorithm is iterative, with one operation scheduled in each iteration [29]. In this version of the algorithm, all the operations are assumed to execute in single control step.

The time frame of each operation is defined as the the control steps in which the operation can be scheduled. As Soon As Possible (ASAP) and As Late As Possible (ALAP) schedules determine the time frames. An ASAP schedule is the one in which every operation is scheduled in the earliest possible time-step allowed by the dependency constraints. On the other hand, ALAP schedule assigns an operation to the latest possible time-step. The mobility of an operation is defined as the difference between its ASAP time-stamp, $T_{ASAP}$ and the ALAP time-stamp, $T_{ALAP}$. The width of time frame of an operation is equal to its mobility plus 1. The operation probability is a function that is zero outside the time frame and is equal to reciprocal of frame width inside it.

Consider the data flow graph presented in 2.3 with a unit delay multiplier and a unit delay adder. From ASAP and ALAP schedules shown in Figure 2.4 and Figure 2.5, operation 1 has zero mobility. Hence, $p_1(1) = 1$ and $p_1(2) = p_1(3) = p_1(4) = 0$. Similar considerations apply to operation 2. Operation 3 has mobility 1 since its time frame is [1,2]. Consequently, $p_3(1) = p_3(2) = 0.5$ and $p_3(3) = p_3(4) = 0$.

The type distribution is the sum of probabilities of the operations that are implementable by a specific resource type from the resource set. The type distribution indicates the concurrency of similar operations in a given time-step. The following equation describes the type distribution.

$$q_k(l) = \sum_{i \in k} p_i(l)$$ (2.5)
where \( q_k(l) \) indicates the type distribution in time-step \( l \) for operation type \( k \). Thus, the type distribution for multiplier (i.e. \( k = 1 \)) in time-step 1 (i.e. \( l = 1 \)), \( q_1(1) = 1 + 1 + 0.5 = 2.5 \). The type distribution for each operation type is calculated for every time-step.

In Force-Directed Scheduling, the decision about assignment of an operation to a time-step is based on the concept of force. The force exerted by an elastic spring is proportional to the displacement of its end points, the spring constant being the proportionality factor. The analogy of mechanical force is applied to the operation assignment by comparing the displacement with the change in probability and spring constant with the type distribution. A force consists of two components - the self force and the predecessor-successor force.

The self force is the sum of the forces relating an operation to all schedule steps in its time frame. The self-force for assigning operation \( i \) with time frame \([l_i^S, l_i^L]\) and mobility \( \mu_i \) to the time-step \( l \) is given as:

\[
self\ force = q_k(l) - \frac{1}{\mu_i + 1} \sum_{m=l_i^S}^{l_i^L} q_k(m) \tag{2.6}
\]

The assignment of one operation to a specific time-step may reduce the time frames of other operations, which are linked with it by dependency relations. The effect on such operations is
considered by predecessor-successor force. Let \([t_i^S, t_i^L]\) be the initial time frame and \([\tilde{t}_i^S, \tilde{t}_i^L]\) be the reduced time frame. Then, following equation gives the predecessor-successor force:

\[
ps\text{-}force = \frac{1}{\mu_i + 1} \sum_{m=\tilde{t}_i^S}^{t_i^L} q_k(m) - \frac{1}{\mu_i + 1} \sum_{m=\tilde{t}_i^S}^{t_i^L} q_k(m)
\]  

(2.7)

The total force for an assignment is calculated as the sum of self-force and all ps-forces.

\[
total\text{-}force_i(l) = self\text{-}force_i(l) + ps\text{-}force_i(l)
\]  

(2.8)
In each iteration of the FDS algorithm, the time frames, probabilities and forces are calculated and the operation with the least force is scheduled in the corresponding time-step. The process is continued until all the operations are scheduled. The computational complexity of the algorithm is cubical in number of operations.

In [12], [11], the authors have presented a modification to the original FDS, referred to as FDS-LP, for dynamic power optimization at the behavioral level. The FDS-LP algorithm optimizes the dynamic power by reducing the switching capacitance inside the resources. The force-directed technique has been used by relating the the dynamic power to the force and establishing the analogies between their parameters. Sharing of the same functional module by two or more similar operations leads to switching activity inside that module. The resulting switched capacitance and the possibility of such a resource sharing combination are evaluated. To model the dynamic power as a force, the switched capacitance due to a resource sharing combination is related to the spring constant, \( k \), and the probability of selecting such a combination is related to the displacement, \( x \), in the force equation \( f = k\cdot x \). Thus, the product of the switched capacitance associated with a resource sharing combination and its selection probability, determine its power cost. This power cost acts as a force associated with the combination.

### 2.5 Our Work

The work presented here describes a new instruction scheduling algorithm based on the classical force-directed approach. A Force-Directed Scheduling (FDS) algorithm was proposed by Paulin and Knight [29] for high-level synthesis. We have modified the FDS to optimize power through instruction scheduling. The advantage of using the FDS is that it tries to find a globally optimized solution. In each iteration of the algorithm, FDS takes into account the global effect of every tentative assignment and thus avoids getting trapped into a locally optimized solution [12]. As seen in the previous section, most of the existing instruction scheduling techniques for power optimization use list scheduling approach. List scheduling heuristic suffers the shortcoming that it can be trapped in local minima of the cost function and therefore may not find globally optimal solution. This motivates us to use the FDS, which promises globally optimal solution, in our work.
It should be noted that our approach for instruction scheduling is based on the force-directed approach, however, the model itself is quite different in terms of its parameter modeling, their role in the force equation and also the level (i.e. compiler-level) at which the model is applied. We model software power as a force for optimization. In our model, the spring constant, \( k \), of the force equation is related to the inter-instruction power cost and the displacement, \( x \), is related to change in instruction probability. Thus, our model differs from the other force-directed models described in the previous section.

2.6 Summary

This chapter describes the existing works in architecture level and software level power estimation and optimization. We also present, in detail, the Force-Directed Scheduling algorithm and steps involved such as time frame computation, probabilities, and force calculations. The FDS technique accounts for the global effect of an assignment during the scheduling process. Hence, FDS provides an advantage of avoiding locally optimized solutions. Also, none of the existing instruction scheduling methods for power optimization at compiler level or software level have used the force-directed approach. This is the motivation behind using a modified FDS to schedule instructions for power optimization at the compiler level, in our work.
CHAPTER 3
LOW POWER FORCE-DIRECTED INSTRUCTION SCHEDULING

We present a new instruction scheduling method for power optimization called Low-Power Force-Directed Instruction Scheduling (FD-ISLP). FD-ISLP is a latency constrained algorithm that reorders the instructions in a basic block so that the total power consumption of the basic block is reduced. Given an assembly code, it is divided into basic blocks and a data dependency graph (DDG) is constructed for each of the basic blocks. SimplePower is used for power characterization of the ISA to obtain a power dissipation table (PDT). The FD-ISLP takes as input, a DDG for the basic block to be scheduled for low power and the PDT. The flow of the power optimization methodology is presented in Figure 1.3.

In each iteration of the algorithm, one instruction is scheduled based on the force value. Power is modeled as a force by relating the inter-instruction power cost as a spring constant and the change in instruction probability as displacement and, in the force equation \( f = k \times x \). The power cost associated with the assignment of an instruction to a particular time-step is determined by the change in probabilities and the inter-instruction costs of the instruction itself and the following instructions whose time frames get reduced. Every instruction that can be scheduled in the current time-step, is tentatively assigned to that time-step and the the power cost is computed for this assignment. Finally, the instruction having the least power cost is selected to be scheduled in that time-step. The process is continued until all the instructions in a given basic block are scheduled.

The proposed approach is based on the FDS algorithm by Paulin and Knight [29], which was explained in detail in Chapter 2.

3.1 Modifications to the FDS Algorithm

The FDS has been typically applied for operation scheduling in high-level synthesis. We apply FDS at the compiler level for reordering instructions for power optimization. Our approach aims
at optimizing the software power, whereas the original FDS targets at the resource optimization. Hence, the parameters of the force equation modeled in algorithm are different from those modeled in the FDS. The mechanical force given in Equation 1.1 is repeated here for clarity.

\[ f = k \times x \]

\[ \text{force} = \text{spring constant} \times \text{displacement} \] (3.1)

The FDS is a resource optimization algorithm, which tries to reduce the resource requirements of a design. Therefore, FDS models the resources as a force to be optimized. Resource optimization is done by increasing the operation concurrency, which is the ability to schedule the similar type of operations in the same control step. We have modeled software power as a force to be optimized. We relate the spring constant \( k \) in the force equation to the inter-instruction power cost. The inter-instruction power cost for a pair of instructions is the amount of power consumed when the two instructions are executed, one after the other. The original FDS relates \( k \) to the type distribution, which is indicative of the concurrency of similar operations for specific resources. The type distribution is the sum of probabilities of the operations that are implemented by a specific resource type from the resource set. We model the displacement \( x \) as the change in instruction probability. The instruction probability can be defined as the probability with which that instruction can be scheduled in a particular time-step, considering uniform probability distribution. The instruction probability of an instruction is a function of its time frame, which is equal to the inverse of length of the time frame, for the time-steps within the time frame and is zero for the other time-steps. The FDS models \( x \) as the change in mobility of operations. The mobility of an operation is defined as the difference between its ASAP time-stamp and the ALAP time-stamp.

### 3.2 ISA Power Characterization

The first step involves profiling the power characteristics of the instruction set architecture, to generate a power dissipation table (PDT). The PDT, for an ISA consisting of \( n \) instructions, is an \( n \times n \) matrix. Each entry PDT\([i,j]\) represents the average power consumed when instruction \( i \) is immediately followed by instruction \( j \). We term this value as the inter-instruction power cost. We use an execution-driven, cycle-accurate RT level power estimation tool, SimplePower tool set[50]
for all the power simulations. SimplePower uses a transition sensitive energy model for in-order 5-stage pipelined datapath. To generate each entry in the PDT, the corresponding instruction pair with a *nop* instruction is repeated 200,000 times in an assembly program. The assembly code is then run through the simulator 10 times to obtain the total energy dissipation. The power cost for the corresponding pair is then averaged from the total. Repeating the same instruction pair instead of using a loop avoids any corruption that could arise due to loop overheads. Figure 3.1 shows the details about the PDT generation.

```assembly
_start:
addu $2,$3,$5  
move $2,$3     
nop            
addu $2,$3,$5  
move $2,$3     
nop            
addu $2,$3,$5  
move $2,$3     
nop            
addu $2,$3,$5  
move $2,$3     
nop            
```

...200,000 times

Figure 3.1. PDT Generation Example

### 3.3 Basic Blocks of Assembly Code

The input, a C language program, is cross-compiled for the SimpleScalar architecture, using the *SimpleScalar* tool set [4], version 2.0. The generated assembly language code is divided into basic blocks. A *basic block* is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The algorithm [1] used for basic block partitioning is presented in Figure 3.2. The proposed approach FD-ISLP is an *intra-basic block* scheduling algorithm that schedules the instructions only within a basic block. Figure 3.3 shows an example of basic blocks for the *Quicksort* benchmark program.
Input: A sequence of three address instructions
Output: A list of basic blocks with each three-address instruction in exactly one basic block

1. First, determine the set of leaders, a leader is the first instruction in each basic block. We use the following rules:

   (a) The first instruction is a leader.
   
   (b) Any instruction that is a target of a conditional or unconditional branch is a leader.
   
   (c) Any instruction that follows a conditional or unconditional branch is a leader.

2. For each leader, its basic block consists of the leader and all the instructions up to but not including the next leader or the end of the program.

Figure 3.2. Basic Block Partitioning Algorithm

3.4 Data Dependency Graph Construction

For each basic block (BB), a data dependency graph (DDG) is constructed representing the dependency relations among the instructions in a given basic block. The DDG representing a basic block is a directed acyclic graph (DAG). Each instruction in a BB is represented as a node in the DDG. And the dependency between a pair of instructions is represented as a directed edge pointing towards the dependent instruction. Figure 3.4 shows a basic block and its DDG representation.

3.5 Low-Power Force-Directed Instruction Scheduling Algorithm

The FD-ISLP algorithm takes two parameters as input, a DDG representing a basic block and the PDT, and returns a power-optimized schedule. A pseudo-code for the algorithm is presented in Figure 3.5. The algorithm starts with finding the ASAP schedule and the ALAP schedule (lines 3-4) for the input DDG. In the ASAP schedule, each instruction is scheduled at the earliest possible time-step under the dependency constraints and in the ALAP schedule, each instruction is scheduled at the latest possible time-step. The latency constraint is determined as the critical path length of the DDG. For every instruction node in the DDG, time frame is computed from the ASAP and ALAP time-stamps (line 7). The instructions having time frame equal to zero are the critical instructions
and have to be scheduled in the specific time-steps. Hence, they are scheduled at their ASAP time-step (line 9). In each iteration of the algorithm, all the unscheduled instructions are considered. Each unscheduled instruction is tentatively assigned to every possible time-step in its time frame, allowed by the data dependencies (line 16). Then, the power cost associated with this assignment is determined by computing the self-force and the predecessor-successor forces (lines 17-18).

The total force for the assignment is computed as the sum of the self-force and the ps-forces (line 19). The force computations are repeated for all the unscheduled instructions for all feasible time-steps. The tentative assignment that has the least force value amongst all, is chosen and the corresponding instruction is scheduled in that time-step (line 23). The ASAP and ALAP schedules are updated and the time frames are recomputed. The above procedure is repeated until all the instructions have been scheduled.

The algorithm can be explained with an example of a DDG graph and its partial schedule shown in Figure 3.6 and Figure 3.7, respectively. In the Figure 3.6, the number pair, \([i,j]\) next to each instruction node in the DDG, represents its ASAP and ALAP time-stamps. e.g. instruction 1 can be scheduled as soon as in the first time-step and as late as in the forth time-step. Similarly, instruction 5 can be scheduled as soon as in the second time-step and as late as in the fifth time-step. As can be seen from the figure, instruction 6 has both of its ASAP and the ALAP time-stamps equal to 6 and
hence, its time frame is equal to 0. Therefore, instruction 6 is a critical instruction and needs to be scheduled in time-step 6.

In Figure 3.7, a number, [k] alongside of instructions 1 and 6 indicates the time-step in which they have been scheduled and the number pair [i,j] next to all other instruction nodes represents the ASAP and the ALAP time-stamps for each one of them. The scheduled instructions are shown with two concentric circles. Thus, instruction 1 and 6 have been assigned to time-steps 1 and 6, respectively, and the ASAP and ALAP time-stamps of all the instructions have been updated. The time frame of instruction 2 is [2,4]. Assuming uniform probability distribution, instruction 2 can be scheduled in any of the time-steps: 2, 3, and 4, with an equal probability of 0.33. If instruction 2 is scheduled in time-step 2, the power cost of the instruction pair (1,2) should be considered (because, instruction 1 has been already scheduled in time-step 1). Similarly, for scheduling instruc-
Input: Data dependency graph, DDG and power dissipation table, PDT
Output: Low power schedule, LP-schedule

(01) Algorithm FD-ISLP(Graph DDG, PDT)
(02) Begin
(03) ASAP-schedule(G)
(04) ALAP-schedule(G)
(05) unscheduled ← total-inst(G)
(06) for each inst ∈ G
(07) Determine-time-frame(inst)
(08) if (inst.time-frame = 0) then
(09) LP-schedule[inst.ASAP] ← inst
(10) unscheduled ← unscheduled - 1
(11) end for
(12) while(unscheduled) do
(13) for each inst ∈ G
(14) if (inst.scheduled = FALSE) then
(15) for each tstep ∈ time-frame(inst)
(16) Make a tentative assignment(inst,tstep)
(17) Compute self-force(inst,tstep)
(18) Compute ps-force(inst,tstep)
(19) total-force(inst,tstep) ← self-force(inst,tstep) + ps-force(inst,tstep)
(20) end for
(21) end for
(22) min-force(inst,tstep) ← minimum-total-force(inst,tstep)
(23) LP-schedule[tstep] ← inst
(24) inst.scheduled ← TRUE
(25) Update-ASAP(G,inst)
(26) Update-ALAP(G,inst)
(27) Compute time frames
(28) unscheduled ← unscheduled - 1
(29) end while
(30) return LP-schedule
(31) End

Figure 3.5. FD-ISLP Algorithm
tion 2 in time-steps 3 and 4, power costs of instruction pairs (3,2) and (5,2) need to be considered, respectively.

Figure 3.6. An Example of a DDG

Figure 3.7. Partial Schedule for the DDG in Figure 3.6

If instruction 2 is tentatively assigned to time-step 2, then the force computations are done as follows: The tentative assignment changes the scheduling probabilities of instruction 2 in time-steps 2, 3 and 4 to 1.0, 0, 0 respectively. The changes in the instruction probabilities are summarized in Table 3.1. Table 3.2 lists all the possible assignments for instruction 2. In the first assignment inter-instruction cost of the pair (1,2) should be considered because instruction 2 follows instruction 1. If
instruction 2 can be assigned in time-step 3, dependency constraints allow only instruction 3 to be scheduled in time-step 2 (because, instruction 1 is already scheduled in time-step 1). In this case, the inter-instruction cost of the pair (3,2) should be considered. Similarly, for the third assignment, inter-instruction cost of the pair (5,2) should be used. From the discussion in Section 4.1, force is modeled as the product of inter-instruction power cost and the change in instruction probabilities. The self-force is the sum of forces relating an instruction to all schedule steps in its time frame. Hence, self-force for the assignment of instruction 2 to the time-step 2 is given as below:

$$\textit{self-force}(2, 2) = \textit{PDT}[1, 2] \times (1 - 0.33) + \textit{PDT}[3, 2] \times (0 - 0.33) + \textit{PDT}[5, 2] \times (0 - 0.33).$$

Similarly, the self-forces for assignment of instruction 2 to the time-steps 3 and 4, are given by the following two equations, respectively:

$$\textit{self-force}(2, 3) = \textit{PDT}[1, 2] \times (0 - 0.33) + \textit{PDT}[3, 2] \times (1 - 0.33) + \textit{PDT}[5, 2] \times (0 - 0.33).$$

$$\textit{self-force}(2, 4) = \textit{PDT}[1, 2] \times (0 - 0.33) + \textit{PDT}[3, 2] \times (0 - 0.33) + \textit{PDT}[5, 2] \times (1 - 0.33).$$

Table 3.1. Probabilities of Instruction 2 for Assignment (2,2)

<table>
<thead>
<tr>
<th>Time-step</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before assignment</td>
<td>0</td>
<td>0.33</td>
<td>0.33</td>
<td>0.33</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>After assignment</td>
<td>0</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Change</td>
<td>0</td>
<td>(1.00 - 0.33)</td>
<td>(0 - 0.33)</td>
<td>(0 - 0.33)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.2. Possible Assignments for Instruction 2

<table>
<thead>
<tr>
<th>Assignment</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment 1</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>Assignment 2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>Assignment 3</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>-</td>
<td>6</td>
</tr>
</tbody>
</table>

The predecessor-successor forces are computed by evaluating the variation on the self-forces of the predecessors/successors due to restrictions on their time frames [23]. Instructions 2 and 4 have the same time frame [2,4]. Therefore, the assignment of instruction 2 to time-step 2 puts restrictions on time frame of its successor, instruction 3. The possible assignments of instruction 3, before the tentative assignment of instruction 2, are listed in Table 3.4. The tentative assignment of instruction 2 to time-step 2 implies the assignment of instruction 3 to time-step 3 or 4. The change in probabilities of instruction 3 are summarized in Table 3.3. In this case, the ps-force consists only of the successor force, because the predecessor of instruction 2, i.e. instruction 1 has already
been scheduled and the assignment does not change its time frame. Therefore, the ps-force for this
assignment is computed as follows:

\[ ps - force(2, 2) \]
\[ = (0 - 0.33) \times PDT[1, 3] + (0.5 - 0.33) \times PDT[2, 3] + (0.5 - 0.33) \times PDT[4, 3] \]
\[ = 0.5 \times (PDT[2, 3] + PDT[4, 3]) - 0.33 \times (PDT[1, 3] + PDT[2, 3] + PDT[4, 3]) \]
(refer to Equation 2.7).

Table 3.3. Probabilities of Instruction 3 for Assignment (2,2)

<table>
<thead>
<tr>
<th>Time-step</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before assignment</td>
<td>0</td>
<td>0.33</td>
<td>0.33</td>
<td>0.33</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>After assignment</td>
<td>0</td>
<td>0</td>
<td>0.50</td>
<td>0.50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Change</td>
<td>0</td>
<td>(0 - 0.33)</td>
<td>(0.50 - 0.33)</td>
<td>(0.50 - 0.33)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.4. Possible Assignments for Instruction 3

<table>
<thead>
<tr>
<th>Time-step</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment 1</td>
<td>1</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>Assignment 2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>Assignment 3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>-</td>
<td>6</td>
</tr>
</tbody>
</table>

The total force associated with the assignment (2,2) is computed as the sum of the self-force and the ps-forces.

\[ total - force(2, 2) = self - force(2, 2) + ps - force(2, 2) \]

Similarly, the force computations are done for all other unscheduled instruction for their feasible time-steps. The tentative assignment with the lowest force is selected and that assignment is finalized. The same procedure is repeated until all the instructions in the given basic block are scheduled.

The intuitive analysis of the algorithm shows that the algorithm promotes such instruction assignments that would result in the least force with their neighbors. Consequently, these assignment result in the least force for the entire basic block leading to power savings.

The time complexity of the proposed approach is \( O(n^3) \), where \( n \) is the number of instruction-nodes in the DDG. The cubical time complexity of the FD-ISLP can be explained as follows: the algorithm schedules one instruction in each iteration and hence goes through \( n \) iterations to schedule the \( n \) instructions. In each iteration, all the unscheduled instructions are considered. And for every
instruction, forces are computed for each feasible time-step, allowed by the data dependencies. The most time-intensive steps in the algorithm are computing the self-forces and the ps-forces.

3.6 Summary

The problem of instruction scheduling for power optimization has been successfully modeled using the force-directed scheduling technique. The software power is modeled as the force to be optimized, by relating the spring constant to the inter-instruction power cost and the displacement to the change in instruction probability. By considering the global effect during each step of the algorithm, FD-ISLP avoids any locally optimal solutions. The effectiveness of our new algorithm is presented in the next chapter.
CHAPTER 4
EXPERIMENTAL RESULTS

This chapter presents detailed experimental results to verify the effectiveness of the FD-ISLP algorithm described in the previous chapter. The implementation details of the proposed algorithm and the benchmark programs used to validate its efficiency are described. The methodology for power optimization approach is presented in Figure 4.1. Each module is separately coded and tested. For integration, the output of each step was made compatible with the next step. e.g. the data dependency graph (DDG) generation module creates an adjacency matrix representation and the next step of scheduling accepts this representation and the power dissipation table (PDT) to generate a low-power schedule. We also present a brief discussion on SimplePower, the tool that we have used for our power simulations.

4.1 Experimental Procedure

Various steps involved in the proposed power optimization approach are discussed below:

4.1.1 Cross-Compilation

The power simulator, SimplePower emulates an integer subset of the SimpleScalar [4] instruction set architecture (SS-ISA). Therefore, the input benchmark C programs are cross-compiled using the ssbig-na-sstrip-gcc compiler to obtain the SimpleScalar (SS) assembly code.

4.1.2 Basic-Block Partitioning and DDG Extraction

The code for parsing the SS assembly code was written in C and it outputs a list of basic blocks for a given assembly language code. The details about basic block partitioning algorithm can be found in Section 3.3. The output from this step is further applied to the next module, which extracts
Figure 4.1. Power Optimization and Validation Framework
the data dependency graph (DDG) from each basic block. The DDG extraction module is also implemented in C and the DDG is represented as an adjacency-matrix data structure.

4.1.3 PDT Generation

A subset of the SS-ISA was required for our simulations. It consists of 15 instructions, which are listed in Table 4.1. This subset of the instruction set architecture entirely covers all the benchmark programs that we have used. The power characterization of the instructions under consideration, is done in this step. The SimplePower tool set was used to obtain the power consumption of assembly programs, each of which represents an instruction-pair. The details about PDT generation are described in Section 3.2. For its force computations, the FD-ISLP algorithm uses the inter-instruction power cost values from the PDT.

<table>
<thead>
<tr>
<th>No.</th>
<th>Instruction</th>
<th>Brief Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>addu</td>
<td>Integer addition unsigned</td>
<td>addu $sp,$sp,24</td>
</tr>
<tr>
<td>2.</td>
<td>subu</td>
<td>Integer subtraction unsigned</td>
<td>subu $sp,$sp,24</td>
</tr>
<tr>
<td>3.</td>
<td>lw</td>
<td>Load word</td>
<td>lw $31,20($)</td>
</tr>
<tr>
<td>4.</td>
<td>sw</td>
<td>Store word</td>
<td>sw $31,20($sp)</td>
</tr>
<tr>
<td>5.</td>
<td>j</td>
<td>Jump to absolute address</td>
<td>j label</td>
</tr>
<tr>
<td>6.</td>
<td>jal</td>
<td>Jump to absolute address and link</td>
<td>jal label</td>
</tr>
<tr>
<td>7.</td>
<td>beq</td>
<td>Branch if equal to 0</td>
<td>beq $2,$0,$L4</td>
</tr>
<tr>
<td>8.</td>
<td>bne</td>
<td>Branch if not equal to 0</td>
<td>$2,$3,$L5</td>
</tr>
<tr>
<td>9.</td>
<td>sll</td>
<td>Shift left logical</td>
<td>sll $2,$3,2</td>
</tr>
<tr>
<td>10.</td>
<td>slt</td>
<td>Set less than</td>
<td>slt $2,$3,$2</td>
</tr>
<tr>
<td>11.</td>
<td>sra</td>
<td>Shift right arithmetic</td>
<td>sra $2,$3,1</td>
</tr>
<tr>
<td>12.</td>
<td>mflo</td>
<td>Move from LO register</td>
<td>mflo $2</td>
</tr>
<tr>
<td>13.</td>
<td>move</td>
<td>Move register contents</td>
<td>move $2,$4</td>
</tr>
<tr>
<td>14.</td>
<td>la</td>
<td></td>
<td>la $4,a</td>
</tr>
<tr>
<td>15.</td>
<td>li</td>
<td></td>
<td>li $5,0x00000064</td>
</tr>
</tbody>
</table>

4.1.4 FD-ISLP

The low power force-directed instruction scheduling algorithm was coded in C. The algorithm schedules each block of the code separately using the DDG and PDT information to optimize its power consumption. The details about scheduling algorithm and the force computations are pre-
sented in Section 3.3. Each scheduled basic block is then written to an output assembly file. Once the entire assembly file is generated, SimplePower is used to simulate its power consumption.

The proposed scheduling technique has been implemented and tested successfully with various C benchmark programs. The benchmark programs used for this purpose are listed in Table 4.2. To prove the effectiveness of our approach, we also find the power consumption of original benchmark programs and compare the power numbers. It was observed that the scheduling technique could save power for all the benchmarks. We also present a comparison of our results with the work of Choi et al. [49]. The comparison shows that our technique outperforms the other technique, for all the cases under consideration. All the experiments were run on a SunSparc Ultra 10 workstation, running SunOS, with 100 MHz processor and 128 MB RAM.

<table>
<thead>
<tr>
<th>No.</th>
<th>Benchmark</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>bsrch.c</td>
<td>Binary search the same $n$ numbers</td>
</tr>
<tr>
<td>2.</td>
<td>bubble.c</td>
<td>Bubble sort $n$ random numbers</td>
</tr>
<tr>
<td>3.</td>
<td>fir.c</td>
<td>Finite impulse response filter</td>
</tr>
<tr>
<td>4.</td>
<td>hanoi.c</td>
<td>Tower of Hanoi for 1 to $n$ disks</td>
</tr>
<tr>
<td>5.</td>
<td>heap.c</td>
<td>Heap sort the same $n$ numbers</td>
</tr>
<tr>
<td>6.</td>
<td>matm.c</td>
<td>$n \times n$ Matrix multiplication</td>
</tr>
<tr>
<td>7.</td>
<td>perm.c</td>
<td>Permutation of $n$ numbers</td>
</tr>
<tr>
<td>8.</td>
<td>queen.c</td>
<td>Find all the solutions for $n$-Queens’ Problem</td>
</tr>
<tr>
<td>9.</td>
<td>quick.c</td>
<td>Quick sort the same $n$ numbers</td>
</tr>
<tr>
<td>10.</td>
<td>rsa.c</td>
<td>RSA cryptographic algorithm</td>
</tr>
</tbody>
</table>

### 4.2 SimplePower

SimplePower is a framework that can be used for evaluating the effect of high-level algorithmic, architectural, and compilation trade-offs on power. It is an execution-driven, cycle-accurate, RT-level power estimation tool. It is based on the architecture of a five-stage pipeline datapath. The instruction set architecture is a subset of the instruction set (the integer part) of SimpleScalar, which is a suite of publicly available tools to simulate modern microprocessors. The five stages of the pipelined datapath are IF (instruction fetch), ID (instruction decode), EXE (execution), MEM (memory access), WB (write back). At each clock cycle, the SimplePower Core simulates the execution of all active instructions and calls the corresponding power estimation interfaces for all
activated functional units. The cache simulator simulates the status of the instruction cache and data cache. The bus simulator snoops the instruction cache address bus, the instruction cache data bus, the cache address bus, and the data cache data bus. It records the total number of access and the number of transitions on these buses.

This tool evaluates the energy considering the system as a whole rather than just as a sum of parts, and it supports both compiler and architectural experimentations. In the energy estimation procedure we have used, the source C code (benchmark.c) is compiled by the SimpleScaler version of gcc (ssbig-na-sstix-gcc), which generates SimpleScaler assembly codes (benchmark.s). The SimpleScaler assembler gas and the loader/linker gld produce SimpleScalar executables that are then loaded into SimplePower main memory and executed by SimplePower. SimplePower provides the total number of cycles in execution, number of transitions in on-chip buses, switch capacitance statistics for each pipeline stage, switch capacitance statistics for different functional units, and the total switch capacitance. All the simulations were performed assuming 0.35 micron technology (simpower35).

4.3 Power Savings

Tables 4.3, 4.4 and 4.5 provide a summary of the results obtained on the benchmark programs. In Table 4.3, the second column indicates the benchmark programs that are provided with the tool set. The third and fourth columns indicate the total power consumption (in units of $nF$) of the original (unscheduled) benchmark code and the FD-ISLP scheduled code, respectively. The last column shows the power savings achieved using our approach. The maximum power savings obtained is 30.87% and the minimum savings is 3.07%, with an average of 12.68% over all the benchmarks. The power consumption readings were obtained by averaging the power simulations for 10 different sets of 100 randomly generated input patterns. For matrix multiplication, the size of the matrix is indicated with a number enclosed in parentheses, alongside the name of the benchmark. e.g. the results in Table 4.3 show that a matrix of size 10x10 was used.

Table 4.4 shows the number of CPU-cycles required to execute the corresponding benchmark program. It can be observed that our algorithm can save CPU cycles of execution for all the bench-
Table 4.3. Power Savings Results for Benchmarks

<table>
<thead>
<tr>
<th>No.</th>
<th>Benchmark</th>
<th>Total Power Unscheduled (nF)</th>
<th>Total Power FD-ISLP (nF)</th>
<th>Power Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Binary search</td>
<td>11.50</td>
<td>10.84</td>
<td>5.74</td>
</tr>
<tr>
<td>2.</td>
<td>Bubble sort</td>
<td>12370.26</td>
<td>11990.49</td>
<td>3.07</td>
</tr>
<tr>
<td>3.</td>
<td>FIR filter</td>
<td>95.73</td>
<td>84.63</td>
<td>11.59</td>
</tr>
<tr>
<td>4.</td>
<td>Hanoi</td>
<td>63449.17</td>
<td>56996.36</td>
<td>10.17</td>
</tr>
<tr>
<td>5.</td>
<td>Heap sort</td>
<td>3599.52</td>
<td>3302.92</td>
<td>8.24</td>
</tr>
<tr>
<td>6.</td>
<td>Matrix multiplication (10)</td>
<td>110.36</td>
<td>97.85</td>
<td>11.34</td>
</tr>
<tr>
<td>7.</td>
<td>Permutation</td>
<td>24010.55</td>
<td>20572.24</td>
<td>14.32</td>
</tr>
<tr>
<td>8.</td>
<td>Queen’s Problem</td>
<td>11924.95</td>
<td>10376.14</td>
<td>12.98</td>
</tr>
<tr>
<td>9.</td>
<td>Quick sort</td>
<td>2078.56</td>
<td>1445.91</td>
<td>30.44</td>
</tr>
<tr>
<td>10.</td>
<td>RSA encryption</td>
<td>6867677.30</td>
<td>5599904.07</td>
<td>18.46</td>
</tr>
</tbody>
</table>

mark programs. Thus, FD-ISLP results in power savings without any performance overhead. The results also support the notion that performance improvement can also lead to power reduction.

Table 4.4. Execution Time of Benchmarks

<table>
<thead>
<tr>
<th>No.</th>
<th>Benchmark</th>
<th>Total Cycles Unscheduled</th>
<th>Total Cycles FD-ISLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Binary search</td>
<td>390</td>
<td>354</td>
</tr>
<tr>
<td>2.</td>
<td>Bubble sort</td>
<td>391398</td>
<td>383230</td>
</tr>
<tr>
<td>3.</td>
<td>FIR filter</td>
<td>3062</td>
<td>2875</td>
</tr>
<tr>
<td>4.</td>
<td>Hanoi</td>
<td>220820</td>
<td>202204</td>
</tr>
<tr>
<td>5.</td>
<td>Heap sort</td>
<td>105380</td>
<td>99170</td>
</tr>
<tr>
<td>6.</td>
<td>Matrix multiplication (10)</td>
<td>4005</td>
<td>3852</td>
</tr>
<tr>
<td>7.</td>
<td>Permutation</td>
<td>751789</td>
<td>670746</td>
</tr>
<tr>
<td>8.</td>
<td>Queen’s Problem</td>
<td>469459</td>
<td>425735</td>
</tr>
<tr>
<td>9.</td>
<td>Quick sort</td>
<td>67543</td>
<td>44754</td>
</tr>
<tr>
<td>10.</td>
<td>RSA encryption</td>
<td>212619184</td>
<td>177855947</td>
</tr>
</tbody>
</table>

It was observed that varying the sizes of the input vectors results in almost the same amount of power savings for the codes scheduled using the FD-ISLP. The experiments were performed with sets of randomly generated 100, 500, 1000 and 5000 input patterns. The results obtained with these experiments are summarized in Table 4.5. For matrix multiplication benchmark, the number enclosed in parentheses indicates the size of the input matrix.

As described earlier, the target machine has a 5-stage pipelined architecture: instruction fetch, instruction decode, execution, memory and write back stage. Therefore, the total power consumption was split into consumption of each of the stages and the savings were computed for each of
Table 4.5. Power Savings for Variable Sized Input Vectors

<table>
<thead>
<tr>
<th>No.</th>
<th>Benchmark</th>
<th>Power Reduction(%) for Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>1.</td>
<td>Binary search</td>
<td>5.74</td>
</tr>
<tr>
<td>2.</td>
<td>Bubble sort</td>
<td>3.07</td>
</tr>
<tr>
<td>3.</td>
<td>Heap sort</td>
<td>8.24</td>
</tr>
<tr>
<td>4.</td>
<td>Matrix multiplication</td>
<td>11.34(10)</td>
</tr>
<tr>
<td>5.</td>
<td>Quick sort</td>
<td>30.44</td>
</tr>
</tbody>
</table>

the stages individually. Table 4.6 shows the breakdown of the total power into its components for the Binary search benchmark. It can be observed that the execution stage of the pipeline consumes the highest amount of energy. Tables 4.7, 4.8 and 4.9 summarize the power savings breakdown for the Quick sort, Matrix multiplication and the Heap sort benchmarks, respectively. The results show that our algorithm could save power in all the stages of the pipeline. This could be explained as follows: in every iteration of the algorithm, it schedules an instruction to a time-step such that this assignment will have the least inter-instruction power cost with its neighbors (both the predecessor and the successor instructions) considering all the possible combinations. This reduces the switching activity over all the stages of the 5-stage pipeline. It could be also observed that the FD-ISLP results in maximum power savings for the execution stage.

Table 4.6. Power Consumption Breakdown for Binary Search Benchmark

<table>
<thead>
<tr>
<th>Input Size</th>
<th>IF(%)</th>
<th>ID(%)</th>
<th>Execution(%)</th>
<th>Mem(%)</th>
<th>WB(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3.68</td>
<td>14.59</td>
<td>40.97</td>
<td>10.20</td>
<td>30.56</td>
</tr>
<tr>
<td>500</td>
<td>3.62</td>
<td>14.47</td>
<td>41.50</td>
<td>10.01</td>
<td>30.39</td>
</tr>
<tr>
<td>1000</td>
<td>3.38</td>
<td>15.71</td>
<td>42.18</td>
<td>9.26</td>
<td>29.48</td>
</tr>
<tr>
<td>5000</td>
<td>3.44</td>
<td>15.73</td>
<td>41.73</td>
<td>9.42</td>
<td>29.69</td>
</tr>
</tbody>
</table>

Table 4.7. Power Savings Breakdown for Quick Sort

<table>
<thead>
<tr>
<th>Input Size</th>
<th>IF(%)</th>
<th>ID(%)</th>
<th>Execution(%)</th>
<th>Mem(%)</th>
<th>WB(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>20.89</td>
<td>29.37</td>
<td>32.54</td>
<td>28.02</td>
<td>30.03</td>
</tr>
<tr>
<td>500</td>
<td>21.45</td>
<td>30.04</td>
<td>32.86</td>
<td>27.59</td>
<td>30.52</td>
</tr>
<tr>
<td>1000</td>
<td>21.26</td>
<td>29.85</td>
<td>32.73</td>
<td>28.61</td>
<td>30.46</td>
</tr>
<tr>
<td>5000</td>
<td>21.79</td>
<td>30.69</td>
<td>33.71</td>
<td>28.86</td>
<td>31.40</td>
</tr>
</tbody>
</table>
Finally, we present the comparison of our results with the most recent work in instruction scheduling, by Choi et al. [49]. Power estimation was done using SimplePower, following the same procedure as described earlier. The results of the comparison are summarized in Table 4.10. It could be observed that our algorithm performs better compared to the other technique, for all the benchmarks under consideration. The average percentage improvement in the power savings is noted to be 20.24%. A comparison of the savings in execution time of the two techniques is presented in Table 4.11. It can be observed that the FD-ISLP schedule takes lesser number of CPU cycles as compared to the schedule generated by Choi et al. The results indicate that the average percentage improvement in the performance speed-up is 26.89%.

Table 4.8. Power Savings Breakdown for Matrix Multiplication

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF(%)</td>
</tr>
<tr>
<td>100</td>
<td>2.82</td>
</tr>
<tr>
<td>500</td>
<td>3.06</td>
</tr>
<tr>
<td>1000</td>
<td>3.22</td>
</tr>
<tr>
<td>5000</td>
<td>3.31</td>
</tr>
</tbody>
</table>

Table 4.9. Power Savings Breakdown for Heap Sort

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF(%)</td>
</tr>
<tr>
<td>100</td>
<td>9.16</td>
</tr>
<tr>
<td>500</td>
<td>9.02</td>
</tr>
<tr>
<td>1000</td>
<td>9.54</td>
</tr>
<tr>
<td>5000</td>
<td>9.36</td>
</tr>
</tbody>
</table>

Table 4.10. Power Savings Comparison

<table>
<thead>
<tr>
<th>No.</th>
<th>Benchmark</th>
<th>Power Consumption ((nF)) scheduled / unscheduled</th>
<th>Percentage Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Bubble sort</td>
<td>11627.65 / 11948.92</td>
<td>11990.49 / 12370.26</td>
</tr>
<tr>
<td>2.</td>
<td>FIR filter</td>
<td>91.89 / 100.44</td>
<td>84.63 / 95.73</td>
</tr>
<tr>
<td>3.</td>
<td>Heap sort</td>
<td>3373.01 / 3615.12</td>
<td>3302.92 / 3599.52</td>
</tr>
<tr>
<td>4.</td>
<td>Quick sort</td>
<td>1919.85 / 2711.44</td>
<td>1445.91 / 2078.56</td>
</tr>
</tbody>
</table>
Table 4.11. Savings in Running Time

<table>
<thead>
<tr>
<th>No.</th>
<th>Benchmark</th>
<th>Execution Time (cycles)</th>
<th>Percentage Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>scheduled / unscheduled</td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>Bubble sort</td>
<td>360675 / 368100</td>
<td>3.47%</td>
</tr>
<tr>
<td>2.</td>
<td>FIR filter</td>
<td>2790 / 2900</td>
<td>61.48%</td>
</tr>
<tr>
<td>3.</td>
<td>Heap sort</td>
<td>96329 / 101136</td>
<td>18.59%</td>
</tr>
<tr>
<td>4.</td>
<td>Quick sort</td>
<td>611170 / 85502</td>
<td>24.00%</td>
</tr>
</tbody>
</table>

4.4 Summary

In this chapter, we have discussed in detail the experimental procedure followed to validate the effectiveness of our algorithm. We have discussed in brief the simulation tool called SimplePower, using which we have performed all the power simulations. Then, we present the results and the power savings as compared to original code as well as the work of Choi et al. From the results, it can be concluded that the problem of instruction scheduling for low power can be successfully formulated using the Force-Directed approach.
CHAPTER 5
CONCLUSIONS

Power optimization has become a desirable feature for computer systems because of several reasons such as high cooling and packaging cost and less reliability, associated with high power consumption. The ability of software to direct much of the activity at the hardware has led to the effort to optimize the software power. We have presented a novel instruction scheduling technique, FD-ISLP, for software power optimization. Our approach of instruction scheduling is based on the Force-Directed approach. We have modeled the power consumption as a force to be optimized by relating the spring constant to the inter-instruction power cost and the displacement to the change in instruction probability. FD-ISLP saves software power without causing any performance overhead. The results show that our technique offers an average of 12.68% reduction in power consumption over the original code for the given benchmark suite.

In the experiments that we performed, we restricted the benchmark programs to those with only the integer instructions subset of the SimpleScalar instruction set architecture, listed in Table 4.1. Further experiments need to be carried out for the entire instruction architecture, including the floating pointing instructions, which requires the power characterization of the of other instructions. Further, the time complexity of the original FDS algorithm proposed in [29], is $O(n^3)$. Recently, a few works towards improving the time complexity of the FDS algorithm have been reported in [46], [47] and [48]. These works need to be investigated and a better implementation in terms of run-time for the FD-ISLP needs to be explored. FD-ISLP is an intra-basic block scheduling algorithm. The effect of inter-basic block scheduling using the force-directed scheduling can also be explored in future.
REFERENCES


