2005

Performance Oriented Scheduling with Power Constraints

Brian C. Hayes
University of South Florida

Performance Oriented Scheduling with Power Constraints

by

Brian C. Hayes

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

Major Professor: N. Ranganathan, Ph.D.
Rafael Perez, Ph.D.
Srinivas Katkoori, Ph.D.

Date of Approval:
March 30, 2005

Keywords: software, reordering, optimization, compiler, assembly

DEDICATION

I would like to dedicate this work to my wonderful family for all of the help and support they have provided throughout my life as well as my loving fiancé, Loren.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Ranganathan for all of his guidance and assistance throughout the course of this work. I appreciate his willingness to become my major professor and provide his knowledge and experience to allow me to succeed. Without him, none of this would have been possible. I would also like to thank Dr. Rafael Perez and Dr. Srinivas Katkoori for serving on my thesis committee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Performance Oriented Scheduling with Power Constraints

Brian C. Hayes

ABSTRACT

Current technology trends continue to increase the power density of modern processors at an exponential rate. The increasing transistor density has significantly impacted cooling and power requirements and if left unchecked, the power barrier will adversely affect performance gains in the near future. In this work, we investigate the problem of instruction reordering for improving both performance and power requirements. Recently, a new scheduling technique, called Forced Directed Instruction Scheduling, or FDIS, has been proposed in the literature for use in high level synthesis as well as instruction reordering [15, 16, 6]. This thesis extends the FDIS algorithm by adding several features such as control instruction handling, register renaming in order to obtain better performance and power reduction. Experimental results indicate that performance improvements up to 24.62% and power reduction up to 23.98% are obtained on a selected set of benchmark programs.

CHAPTER 1

INTRODUCTION

Moore's Law, which has correctly predicted the exponential performance gains seen in the realm of computer architecture, currently predicts the number of transistors on a single integrated chip will double every 18 months. This prediction has held for almost 30 years. With ever increasing transistor density, power density is also increasing (see Figure 1.1). Large power densities not only degrades integrated circuits, it also wastes precious power, especially in portable applications where battery technology has not increased at the same rate. In addition, the cost to cool such integrated circuits becomes exponentially expensive, as evidenced in Figure 1.2. Until recently, many approaches still maintained performance as the primary design objective, with a secondary consideration to power consumption. However, it is now realized that power must also become a primary objective in order to maintain performance.

As the technology field begins to create faster and smaller integrated circuits, the problem of power, heat, and cooling will become more problematic. Therefore, new techniques must be developed to reduce power consumption, while still maintaining performance. Current research now focuses on finding ways to reduce power at all levels, from the hardware to the software. In industry, a great deal of effort has been invested in hardware research in order to design for power. As a result, many power efficient designs have been developed. However, this optimization alone will not solve the power issue. Software architects/programmers traditionally have not spent a great deal of time in attempting to create efficient code and certainly have not addressed how to reduce power. Therefore, there is still a great deal of work in the software area in the realm of power reduction. Although there is much to be done in all areas, power aware software lags greatly when compared to its hardware counterparts. This paper intends to provide one additional technique at the

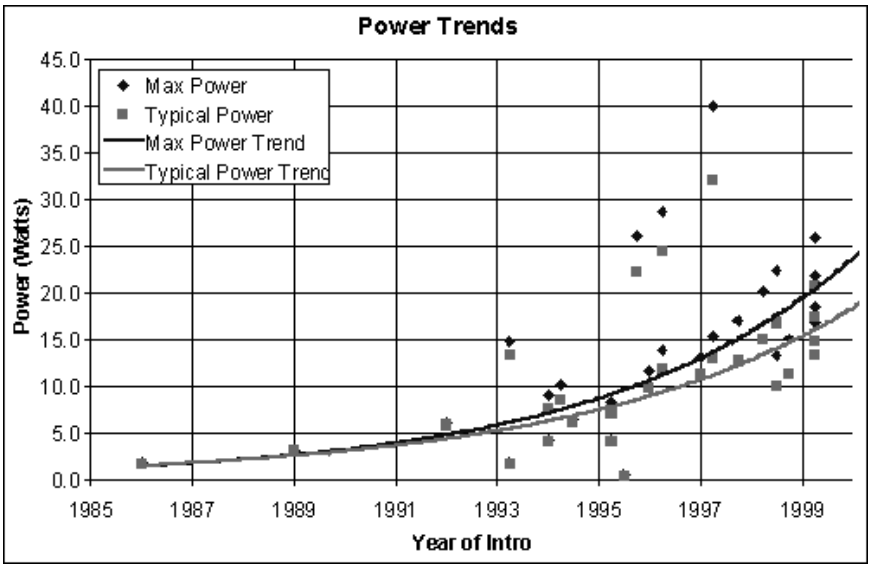

Figure 1.1. CPU Power Trends from 1985-1999[19]

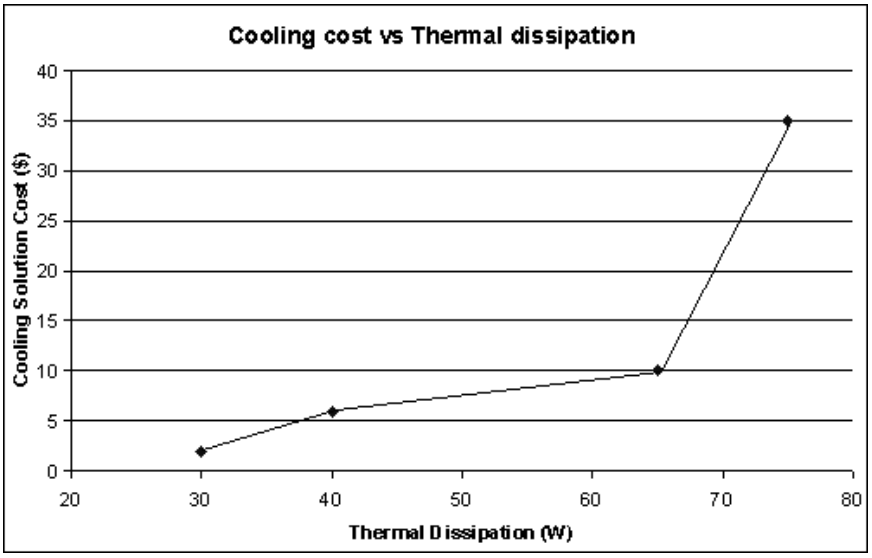

Figure 1.2. CPU Cooling Cost vs. Power Dissipation[19]

software level to assist in addressing this imminent power problem as well as introduce an additional performance gain.

The force directed scheduling based algorithm presented in this paper specifically addresses the issue of reducing the amount of power needed to execute a given program. However, the algorithm's primary objective is to improve performance while gaining a power savings. Force directed scheduling has been successfully applied in hardware design and is one of the primary techniques used in the field. However, few attempts have been made to apply this method in software scheduling. The FDIS algorithm works by using the inter-instruction power cost as a scheduling metric. Specifically, the algorithm attempts to minimize the power utilized by the CPU to switch from one instruction to the next successive instruction. Before scheduling, the algorithm segments the assembly level code into basic blocks, upon which the scheduler can act. For each basic block (defined as a segment of code executed in sequence without jumps or branches), the algorithm attempts to reorder the instructions within a basic block such that the cost to switch between instructions is minimized. Since the data dependency graph and critical path of the program is calculated ahead of time, the algorithm specifically excludes those instructions along the critical path from being rescheduled so that the performance of the program is not negatively affected. Therefore, the algorithm produces a new version of the code that runs no longer than the original version and ideally, uses no more power than the original code. However, since the algorithm is only locally optimized, these statements cannot be theoretically proved, but have thus far held under all experiments performed. In general, most instructions are not contained within the critical path and therefore provide a significant opportunity minimize power.

Using the techniques and algorithms developed in this paper, average performance gains of 2.8% were seen, with a maximum of over 26%. In addition, power savings in excess of 3% was also achieved, with as much as 24% attained. These results are significant in that this technique does not preclude the use of other, previously developed techniques. Furthermore, the operations upon the source can be performed quickly and efficiently.

1.1 Thesis Organization

The organization of the paper is as follows. Section 2 contains the related work, including a through description of the two major bodies of work which this paper uses to validate these results. Section 3 describes the actual algorithm and a discussion of how the results were to be obtained. Section 4 contains the actual results of the experiments performed and the analysis of these results. Finally, section 5 summarizes the results and provides a basis of future research.

CHAPTER 2

RELATED WORK

Over the past 10-15 years, there have been many techniques and approaches to reduce power consumption at all levels. However, it has been a relative recent phenomenon that so much effort is being placed into this area of research, as power and heat are the two main obstacles to face within the next 10 years.

2.1 SimpleScalar

SimpleScalar was written in 1992 as part of the Multiscalar project at the University of Wisconsin[21]. As seen in Figure 2.1, the SimpleScalar simulation suite contains the ability to simulate program binaries using a wide range of simulations. In this work, the SimpleScalar framework was used to translate the benchmarks into assembly level code to be simulated in the SimplePower framework.

2.2 SimplePower

SimplePower is an execution-driven, cycle-accurate RT level energy estimation tool that uses transition sensitive energy models as the basic framework. SimplePower can also provide the energy consumed in the memory system and on-chip buses using analytical energy models[22]. SimplePower is one of the few tools to estimate the power consumption at the RT level. One of the downsides to the SimplePower tool set is that it only works on the integer subset of the SimpleScalar tool set, which limits the code that can be simulated. The simulator estimates the switch capacitance of the processor data path, memory, and on-chip buses[22]. The only items that are not estimated are the control unit, the clock generator, and the distribution network. SimplePower consists of 5 main components: the SimplePower core, the RTL power estimation interface, the technology dependent switch

SimpleScalar baseline simulator models			
Simulator	*Description*	*Lines of Code*	*Simulation speed*
Sim-safe	Simple function simulator	320	6 MIPS
Sim-fast	Speed-optimized functional simulator	780	7 MIPS
Sim-profile	Dynamic program analyzer	1,300	4 MIPS
Sim-bpred	Branch predictor simulator	1,200	5 MIPS
Sim-cache	Multilevel cache memory simulator	1,400	4 MIPS
Sim-fuzz	Random instruction generator and tester	2,300	2 MIPS
Sim-outorder	Detailed microarchitectural model	3,900	0.3 MIPS

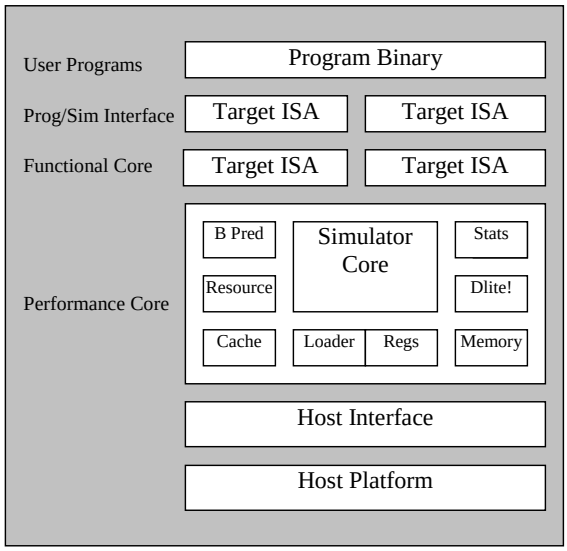

Figure 2.1. SimpleScalar Simulators and SimpleScalar Organization[21]

Table 2.1. Register Renaming Power Savings in Memory

Benchmark	Original (nF)	Relabeled (nF)	Reduction (nF)
acker.c	17,742.8	15,784.5	11.04%
bsrch.c	0.936	0.843	9.94%
bubble.c	1,033.8	870.5	15.79%
fib.c	166,117.1	146,017.5	12.10%
hanoi.c	690.9	600.7	13.05%
heap.c	338.6	302.7	10.61%
matmult.c	10,075	8.372	16.90%
perm.c	2,459.6	2,284.5	7.12%
queens.c	1,398.7	1,306.0	6.62%
quick.c	194.8	168.6	13.47%
sieve.c	1,902.2	1,659.5	12.76%
Average	-	-	11.76%
Std. Deviation	-	-	3.04%

capacitance tables, the cache/bus simulator, and the loader. Besides the optional outputs, SimplePower provides the register file final status, the total number of cycles in execution, the number of transitions in the buses, the switch capacitance statistics for each pipeline stage, the switch capacitance statistics for different functional units, and the total switch capacitance[22]. Comparing each of the functional units models to HSPICE, the average error was found to be within 15% for all the units[22].

The simulator, as described above, simulates code compiled, assembled, and linked using the SimpleScalar tools gcc (compiler), gas (assembler), and gld (linker/loader), respectively. The SimplePower framework also contains utilities that actually contain power reduction techniques not found in the SimpleScalar tool set. Specifically, the SimplePower compiler outputs a register renaming mapping during the simulation process that can be used to implement register renaming. Register renaming is an accepted power reduction technique and is incorporated often times in hardware. However, this technique implemented in software can give systems without this hardware feature the power reduction savings. According to [22], this particular implementation of register relabeling actually can save on average 12% in the memory system (see Table 2.1)

2.3 Wattch

In 2000, Brooks, et. al. presented a new type of simulator[4], similar in nature to SimplePower. This simulator, like SimplePower, is based off of the SimpleScalar framework and is cycle accurate. The goal of the simulator is to provide the accuracy of low-level power simulators, like QuickPower and PowerMil, but provide high-level functionality so that

different power architectures could be evaluated without being fully implemented. Wattch can model various types of functional units within a CPU, including register files, queues, caches, memories, reorder buffers, and branch predictors, among others. Unlike SimplePower, Wattch also models clock generation, including clock wiring, capacitive loads, and clock buffers. In addition, Wattch can model hardware as well as software optimizations, so that a complete overview can be obtained. Microarchitectural trade offs, compiler optimizations and hardware optimizations can all be evaluated using this simulator, as seen in Figure 2.2. According to [4], Wattch is accurate to within 10% of other low-level power simulators, yet it is orders of magnitude faster than its low-level counterparts.

2.4 General Software Power Reduction Techniques

While SimpleScalar, SimplePower, and FDS all provide foundation for the algorithm presented in this paper, many other techniques using other approaches can currently being developed to solve the power problem. In terms of software approaches, Tiwari[12] outlined three main techniques that software can use to reduce power consumption. They include: reducing switching activity, generating code using pattern matching, and reducing memory accesses.

2.4.1 Reducing Switching Activity

Reducing switching activity is the one of the primary techniques upon which FDIS works, as FDIS attempts to reduce the inter-instruction power cost. This inter-instruction power cost is related to the previous instruction and the current instruction being executed, which are the same variables that affect switching activity. By minimizing the number of transistor states changes, the power lost due to switching activity can also be minimized.

2.4.2 Pattern Matching

Pattern matching is performed by assigning costs to specific code patterns and then covering the entire program with the code patterns, while minimizing the overall cost. This forces the compiler to utilize efficient code patterns that exist within the program,

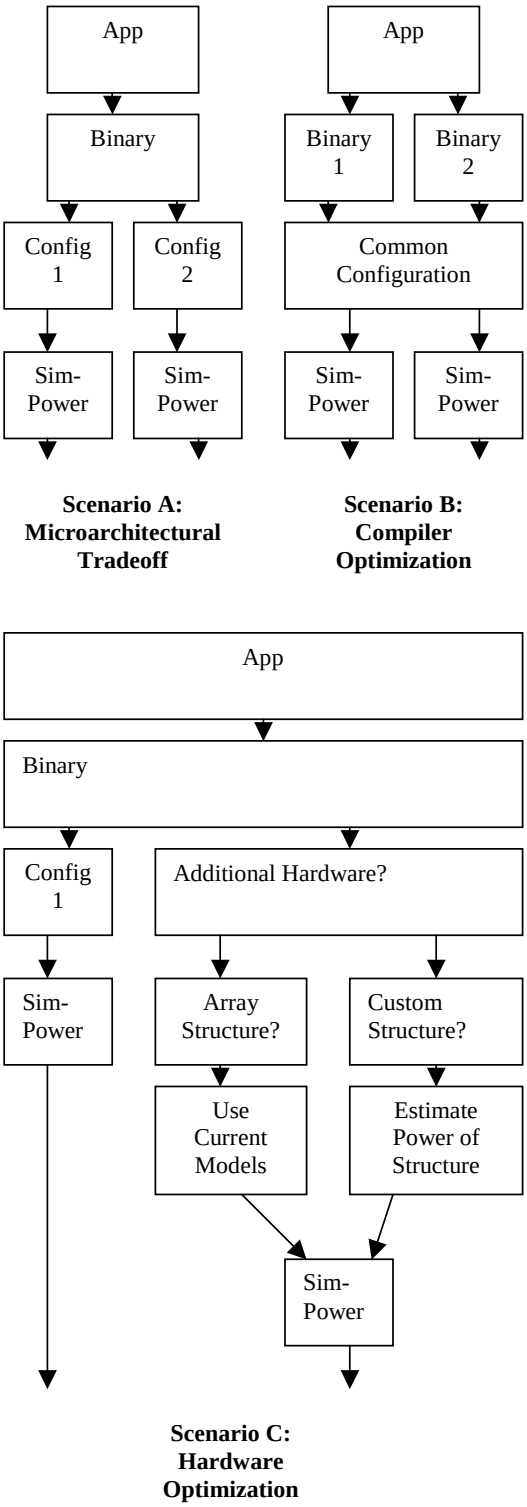

Figure 2.2. Power Analysis Scenarios for Wattch[4]

but are not recognized in a sequential compile of a user-defined program. In addition, this technique may reduce the requirements placed on the programmer to develop power efficient source code.

2.4.3 Memory Access Reduction

The final area of software power reduction is the reduction of memory accesses. This is one area where software optimizations can actually have a greater impact than those optimizations performed in hardware. In [12], an optimized memory system showed a 4% decrease in power when optimized in hardware, but 23-62% decrease when software was optimized. In the realm of overall power, this can be significant. In the 486DX2 processor, register accesses had a power cost of approximately 300mA[12]. Memory reads, on the other hand, have a power cost of nearly 430mA, while memory writes take almost 530mA[12]. Memory writes in the 486DX2 processor also incur an additional power cost since the system utilized a write-through cache coherency model and therefore, a write could occur in every level of the memory hierarchy. These additional writes could significantly increase the cost of write operations and are simply unavoidable in the hardware. Therefore, in cases where hardware optimizations have little effect, software optimizations, which have also been proven beneficial in [10], can greatly reduce overall power consumption.

2.5 Instruction Buffering and Compiler Optimization

In [13], the authors present a technique of inserting a mini-cache between the CPU and the main cache. According to[13], the data and instruction cache is one of the primary power consumers of the CPU. In [5] and [20], it was determined that the instruction cache in the StrongARM SA-110 processor from DEC consumed 27% of the total power dissipated, and even in the Pentium Pro processor, the cache still consumed 14%. In general, memory systems consume a great deal of power to store require data. Caches, because of the large amount of data throughput, also have a great deal of switching activity, as data enters and leaves the cache to go to and from the CPU and memory. The goal of [13] is to reduce the switching activity and extraneous instruction fetching to reduce power

consumption. Others have developed similar techniques, such as a filter cache in [9] and a loop buffer in [17], to reduce power. [13] also introduces another cache, referred to as L-cache, which is accessible by the compiler. As a result, a smart compiler can analyze the code and inform the hardware what to load into the L-cache to further reduce power. Such software/hardware communication may be the approach taken in the future to maximize the power reduction.

2.5.1 Hybrid Approaches

While many techniques work at the hardware or software level, some techniques work at both levels. Many architectures contain parallel mechanisms, such as an Tomasulo implementation, to increase performance of a system. In Tomasulo, there are many extra registers and data paths that consume a large amount of power. One idea presented in [11] is to have a hybrid paradigm, where parallelism that can be seen at the compiler level is implemented in software, allowing hardware parallel schemes to be put to sleep during the execution of compiler optimized code, and only utilizing hardware implemented parallelism where necessary. Although performance becomes a concern, [1] demonstrated that if done properly, power can be reduced while maintaining performance when simply optimizing at the software level.

2.5.2 VLIW

Very Long Instruction Words, or VLIW, is an attempt to minimize power by changing the instruction set architecture to allow for a single fetch that retrieves multiple instructions, as opposed to using one fetch per instruction. The principle behind this idea is to reduce the number of times the memory is accessed and the compiler can package instructions in the most efficient method possible, using power optimization techniques. However, this concept also leads to a more complex control unit and compiler, as well as the need for a larger bus width and registers. Finally, the portability of such a scheme is very low, since the number of instructions packaged and how they are package varies from architecture to architecture. Although a novel idea, the practicality of VLIW has significantly reduced any

attempts to implement such a scheme into any such system. [3] and [2] has demonstrated the theoretical benefits having a VLIW architecture with power optimization, but to date, the refinement has remained in the research community.

2.6 Other Miscellaneous Techniques

In general, higher frequencies result in more switching activity and switch capacitances. Therefore, separate from the explosion in the number of transistors, the burst in frequency has also lead to increase power dissipation. In addition, higher voltage leads to a reduction in delay, in relation to a lower voltage system. Performance can be expressed as directly proportional to the frequency and the voltage. Power is also directly proportional to voltage and frequency. Many research facilities, including Intel, are attempting to find ways to reduce voltage and/or frequency whenever performance is not a top priority. For example, composing a letter in a word processor does not require the fastest processor. Therefore, the frequency and voltage could be reduced, as there would be a significant power savings without a visible decrease in performance.

2.7 Classical Force Directed Scheduling

Scheduling, in particular, is a difficult problem to solve. Many areas, such as design automation and optimization, require scheduling in order to efficiently and effectively accomplish the required tasks. In 1987, Paulin and Knight[15] introduced a scheduling technique for automatic data path synthesis called force directed scheduling (FDS), based on the principle of a spring (see equation 2.1).

$$F = K * x \tag{2.1}$$

Later, in 1989[16], Paulin and Knight introduced an updated version of the technique for behavioral synthesis of application specific integrated circuits (ASICs). Since then, there have been many improvements and updates made to the algorithm in general. Today, FDS is one of the primary scheduling algorithms in behavioral synthesis and has expanded to

many areas such as dynamic power optimization[8] and with the introduction of FDIS, software power reduction.

The ability of the algorithm to be applied to so many areas lies in the inherent trait of the basic blocks upon which FDS acts. FDS can be used in virtually any application upon which there is a need to schedule many basic blocks, subject to a dependency graph. In the case of design automation, FDS is particularly well suited for the design process. The design automation process begins with a description of a circuit usually specified in the form of a hardware description language, such as Verilog or VHDL. From there, a control dependency graph (CDG) and a data dependency graph (DDG) can be constructed. It is at this point where operations (either single or multiple) are mapped to nodes within the two graphs and become the basic blocks that the FDS algorithm will schedule. After mapping the basic blocks to the type of functional unit needed, the algorithm can use the dependency graphs to iteratively schedule the functional units onto the physical units and create the data paths as specified in the graphs.

Scheduling the basic blocks is one of the most difficult tasks, as virtually all basic blocks are connected to other basic blocks; few are ever isolated. This dependency must be realized and maintained during running of the algorithm. Physical constraints, such as the number of wires to connect the basic blocks, must also be taken into account for a realistic schedule that can map to an actual chip. Furthermore, performance is also an item that FDS attempts to address. All units connected together should be as close together as possible to reduce delays and to minimize routing resources used.

To accomplish the task, FDS works in the following manner. Once the data dependency graphs have been created and time frames have been assigned (see Figure 2.3), the ASAP (as soon as possible) (see Figure 2.4) and the ALAP (as late as possible) (see Figure 2.5) schedules can be created. This is done using the critical path of the graphs as the latency restrictor and then calculating, for each operation/basic block, the earliest and latest start times. An operation can be assigned to any of the time frames between the operation's assigned time frame in the ASAP schedule and its time frame in the ALAP schedule. Therefore, the probability of an operation being scheduled outside the ASAP and ALAP

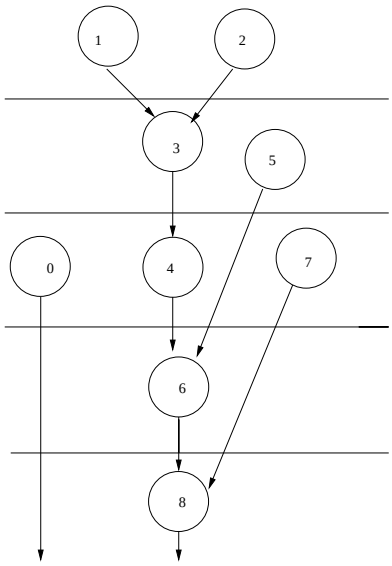

Figure 2.3. Initial Schedule with Assigned Time Frames

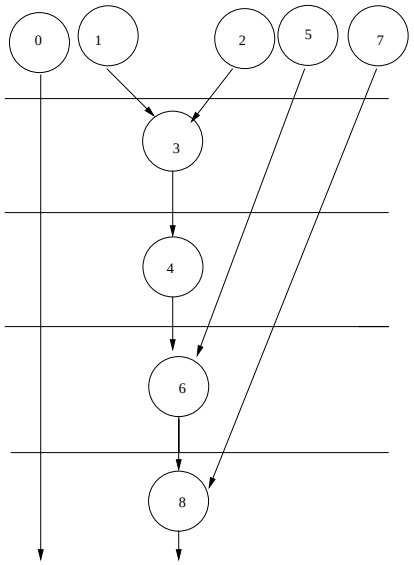

Figure 2.4. As Soon As Possible Schedule

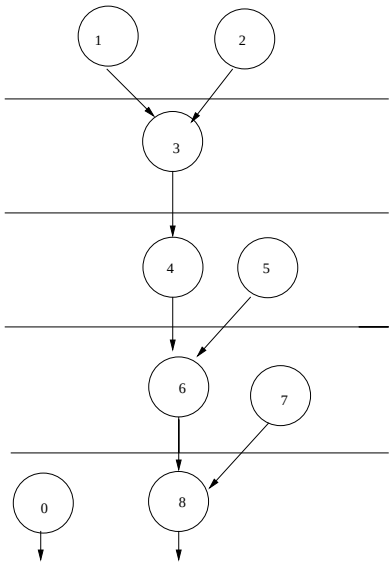

Figure 2.5. As Late As Possible Schedule

schedule is zero and the probability for any time step between the two schedules is equal to:

$$Prob(Operation) = \frac{1}{ALAP - ASAP + 1} \tag{2.2}$$

assuming uniform probability.

Taking the summation of the probabilities of each operation type for a given time step i will result in a distribution graph that reflect the concurrency of a particular operation type at that point in time (see equation 2.3). This value is important for future calculations, as DG(i) is used as the spring constant in the calculation of forces.

$$DG(i) = \sum_{OpnType} Prob(Operation, i) \tag{2.3}$$

For each operation, the force required to change the probability of the operation at a particular time step is given by

$$Force(i) = DG(i) * x(i) \tag{2.4}$$

where x(i) is equal to the change in the operation's probability. Self-force is the force required to assign an operation to a particular time step j. Therefore, this is calculated by summing the force required to change the probability over all time steps, as given by equation 2.5:

$$self - Force(j) = \sum_{i=t}^{b} [Force(i)] \tag{2.5}$$

where the time frame for the specified operation falls between t and b.

The final step of the FDS algorithm before scheduling an operation is to determine the predecessor and successor forces. Whenever an operation is scheduled to a particular time step, the possibilities for other operations are reduced and thereby affect the probabilities of the remaining operations. Therefore, the force required to change the probabilities of these operations should also be added to the self force of the operation to determine the total force, as given by equation 2.6:

$$Total - Force = self - force + ps - force \tag{2.6}$$

Overall, the summary of the force directed scheduling algorithm is as follows:

Repeat until all scheduled

Find ASAP schedule

Find ALAP schedule

Calculate DG(i), using Equation 2.3

Calculate Self(j), using Equation 2.5

Find Total-Force, using Equation 2.6

Schedule Op w/ lowest force

End Repeat

2.8 FD-ISLP

In 2003, Dongale[6], developed an algorithm based on force directed scheduling for creating power optimized code. The algorithm called Force Directed Instruction Scheduling

for Low Power, or FD-ISLP, was proposed, however, several issues were not addressed and thus the results were not reflective of the real picture. In this work, a basic block is defined as a segment of code which contains no branch or jump instructions, with the exception of the last instruction. During the scheduling phase, jump or branch instructions must remain as the last instruction. Otherwise, the semantics of the code is compromised. While there was a provision to ensure such instructions did not move within the FD-ISLP algorithm, the developed mechanism did not guarantee that the branch instructions were not moved. As a result, the schedule may result in some code segments not being executed. The algorithm presented in this work, called FDIS is a modified version of FD-ISLP that has corrected the scheduling error, as well as provided additional improvements such as register renaming to the scheduling paradigm, providing better performance and power reduction.

In addition to correcting the scheduling error within the FD-ISLP algorithm, other significant changes were also made. Most notably was the inclusion of register renaming. This well known technique can be successfully incorporated into the algorithm and provide a relatively significant performance boost. Furthermore, an operand power table was also included into the algorithm to provide a more complete power characterization. In the original FD-ISLP algorithm, power values were attained for the instructions only, without any consideration for the operands. Therefore, two instructions of the same type would be given preference over two instructions with the same operands. Although, this may generally prove to be correct, such an assumption cannot be made. Therefore, a power analysis was performed and incorporated into the updated algorithm to fully characterize an instruction. While no significant effect on the overall performance was seen in the final results, as explained in section 3.2.1, such an update may prove useful in specialized cases. For example, if a basic block contained nothing but a single instruction type (i.e. a list of add instructions), the FD-ISLP algorithm would calculate the same power cost for all of the instructions and would be forced to keep the original schedule. However, in FDIS, the operands can be considered in order to take advantage of the same operand appearing in multiple instructions. An important analysis we provide is that force directed methods result in clock cycle reduction and thus result as performance improving algorithms and

the power savings depend on the reduction in the number of clock cycles besides the inter-instruction and common operand detection for power optimization.

CHAPTER 3

FORCE DIRECTED INSTRUCTION SCHEDULING

Conceptually, the technique used in this paper is quite similar to the classical version of force directed scheduling. Instead of basic blocks consisting of circuits to be mapped to functional units, our basic blocks consist of instructions that are mapped to functional units within the computer, such as adders, multipliers, and dividers. Essentially, the algorithm developed is a level of abstraction above the classical version of force directed scheduling. Rather than determining the schedule for placement on an ASIC, this algorithm determines the schedule of execution for instructions on a given instruction set architecture.

3.1 FDIS vs. FDS

The primary difference between the two algorithms is how the self and successor forces are calculated. Whereas in force directed scheduling the primary metric for optimization is concurrency of operations, the goal of FDIS is power reduction and performance enhancement. Since the power needed to execute an instruction is difficult to reduce, the concept behind FDIS is to reduce the power cost associated with moving from the current instruction to the next sequential instruction. For example, switching from an add operation to a multiply operation may require a significant amount of switching activity[1] on the data path in order to begin the multiply instruction, because of the required accesses. However, switching from one add operation to another, with only one operand changing, will require less power. Therefore, in FDIS, the "spring constant" is the cost of moving an instruction from one time frame to the next, in relation to the inter-instruction power increase or decrease.

[1]The power dissipated from this switching activity will be referred to as the inter-instruction power cost.

Table 3.1. Sample Power Dissipation Table

Inst.	addu	addi	beq	bgtz	bne	j	jal	la
addu	0	6	7	5	3	5	6	2
addi	6	0	0	2	3	7	5	9
beq	7	0	0	5	0	3	6	1
bgtz	5	2	5	0	4	5	2	5
bne	3	3	0	4	0	0	6	8
j	5	7	3	5	0	0	6	1
jal	6	5	6	2	6	6	0	7
la	2	9	1	5	8	1	7	0

3.2 Power Characterization

Inter-instruction power could vary from one computer to the next, depending on the computer architecture and organization. The power values are not easily calculated, but the relationships between instructions must be known to the FDIS algorithm in order to accurately determine the self and successor forces. Therefore, the algorithm not only requires the source code, but also a matrix relating the inter-instruction power cost for all instructions types.

In Table 3.1, a sample power dissipation table, or PDT, is given. It can be seen that the actual power cost associated between two instructions is not important. Only the relationship between the two instructions is needed. For example, from Table 3.1, switching from an *addi* to an *addu* instruction has a cost factor of 6, while the cost factor between an *addi* and a *bne* instruction is 3. Therefore, the *addu* instruction transition will have inter-instruction power cost twice that of a *bne* transition when the preceding instruction is *addi*. Note that the table is symmetric, indicating that for this particular computer organization, simply reversing the order of instructions does not yield a power savings.

A subtle requirement for the algorithm is to know the entire instruction set of the source program and that the entire instruction set is characterized in the PDT. Otherwise, the algorithm may encounter an unknown instruction or may not be able to calculate all of the self and successor forces for all of the instructions. However, the characterization only needs to be performed once for a particular instruction set architecture and associated organization. Once the characterization is complete, it can be stored and reused for future

iterations of the algorithm. This inherent trait of the algorithm also allows for theoretical analysis, as the characterization can easily be modified to adapt to a theoretical model of a computer, assuming the characterization can be determined.

3.2.1 Operand Power Table

Overall, the power dissipation table does not fully characterize an instruction. Instead, the PDT simply characterizes the operation. That is, it does not consider the operands. While the PDT considers the switching activity that occurs as a result of control lines, no data is collected about switching activity on the address lines. In order to more fully characterize all instructions, an operand power table was created using a similar means used to generate the PDT. While the PDT used the same operands and simply changed the operation, the operand power table was developed by selecting a single operation, such an *add* instruction, and changing the operands to see the overall affect. Since the SimpleScalar architecture contains 32 registers, a 32x32 matrix was developed to characterize the cost of switching from one operand to the next. The concept behind this approach was to sum the PDT value, as well as the cost value of switching from one register to the next[2]. The percentage of the total switching activity that the address lines use was unknown so experimental runs were performed to find what fractional weight gave an increase power reduction or performance gain. For example, the operand value could be given a weight of 25% so that the PDT value would carry 4 times the weight. This was done because it was believed that the operation switching activity is the primary cost factor. In reality, it was found that the operation power value is the dominate power value, as all attempts to incorporate an operand power value either had no effect on the overall schedule or negatively impacted the schedule. As a result, this scheme was not incorporated into the final version of the algorithm.

[2]For a given instruction, there can be up to 3 registers. Therefore, the algorithm could potentially sum the PDT value with 3 operand cost values (one for each register location)

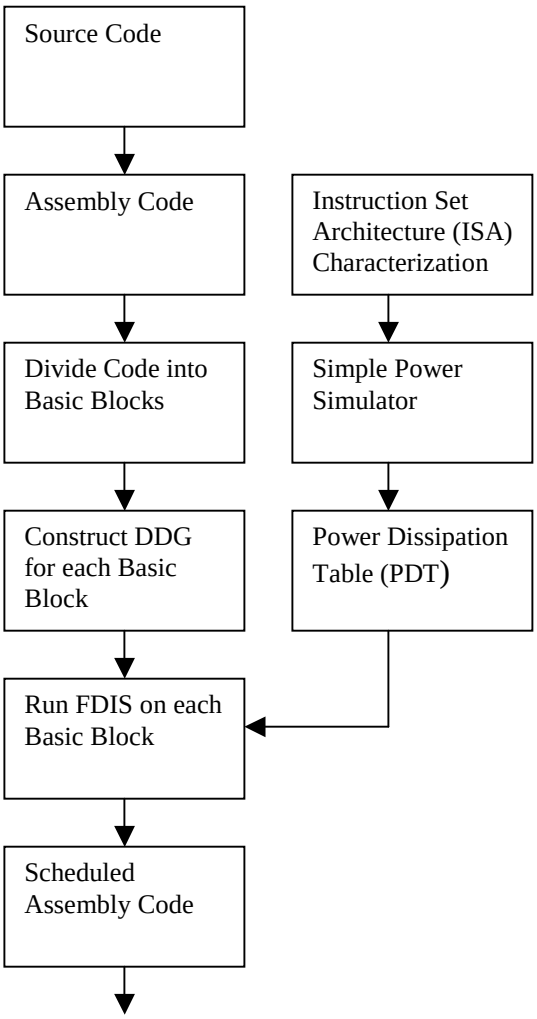

Figure 3.1. Block Diagram of FDIS Algorithm

3.3 FDIS Algorithm

The following is a pseudo-code version the FDIS algorithm[3]. As mentioned before, the algorithm requires the assembly level source code and the power dissipation table for the desired computer organization. Figure 3.1 is a block diagram of the algorithm.

FDIS(Source Code, PDT)

(01) Segment code into basic blocks

(02) For each basic block

(03) Create data dependency graph (DDG)

[3]It is assumed that the algorithm is able to determine the instruction set from the PDT

```
(04)        Unscheduled = number of instructions
(05)        Find ASAP Schedule
(06)        Find ALAP Schedule
(07)        For all instructions
(08)            If ALAP-ASAP == 0
(09)                Schedule instruction
(10)                Unscheduled–
(11)            End If
(12)        End For
(13)        While Unscheduled != 0
(14)            Update ASAP Schedule
(15)            Update ALAP Schedule
(15)            For all unscheduled instructions
(16)                For all time steps
(17)                    Assume Inst. assigned to time step
(18)                    Calculate self Force using PDT
(19)                    Calculate successor forces using PDT
(20)                    Calculate total force
(21)                End For
(22)        End For
(23)        Find instruction with minimum force
(24)        Schedule instruction
(25)        Unscheduled–
(26)        End While
(27)    End For
Return scheduled source code
```

The original force directed scheduling algorithm can be categorized into five steps or stages. It can be observed that many of these same steps are completed in FDIS. In FDS, the first stage is to calculate the ASAP and ALAP schedules. Using the critical path as a guide, each instruction has an earliest and latest time that it can be scheduled in order

for the program to complete on time. Therefore, in lines 5 and 6 of the FDIS algorithm, the ASAP and ALAP schedules are determined. Step 2 updates the distribution graph so that the self and successor forces can be calculated in steps 3 and 4. Since the distribution graph is not vitally important to the implementation of FDIS, these three steps are merged and completed in lines 7-22. Step 5 consists of scheduling the operation with lowest force to the selected time frame. Similarly, FDIS performs this operation in lines 23-24. Finally, both FDS and FDIS loop until all operations/instructions as scheduled. Lines 25-27 handle this loop overhead.

3.4 Basic Blocks

The FDIS algorithm works upon basic blocks only. A basic block is defined as a block of sequential instructions where no breaks or branches occur within the code segment. The only exception is that a break or branch instruction may occur as the last instruction of the code segment. Therefore, the first task of the algorithm is to divide the code into basic blocks, upon which the algorithm will operate. Figure 3.2 is a sample segmentation of source code.

For a given basic block, a data dependency graph can be constructed to reflect the dependencies within that basic block. Since there is only one entry and exit point from any given basic block, as long as the data dependencies within that basic block are maintained, correct code execution will result, regardless of the final schedule.

3.4.1 Inter-Block Scheduling

Due to the implementation style of the algorithm, it is clear that intra-block schedules will be optimized. However, this does not guarantee a globally optimal schedule. Theoretically, it may be possible to change the order of execution of two or more basic blocks and create an even more optimal schedule. Creating a globally optimal schedule using FDIS would require the algorithm to not only create optimal intra-block schedules, but also inter-block schedules. Such an idea, in practice however, is not possible with FDIS. In order to use FDIS to schedule basic blocks, an inter-block schedule would require another

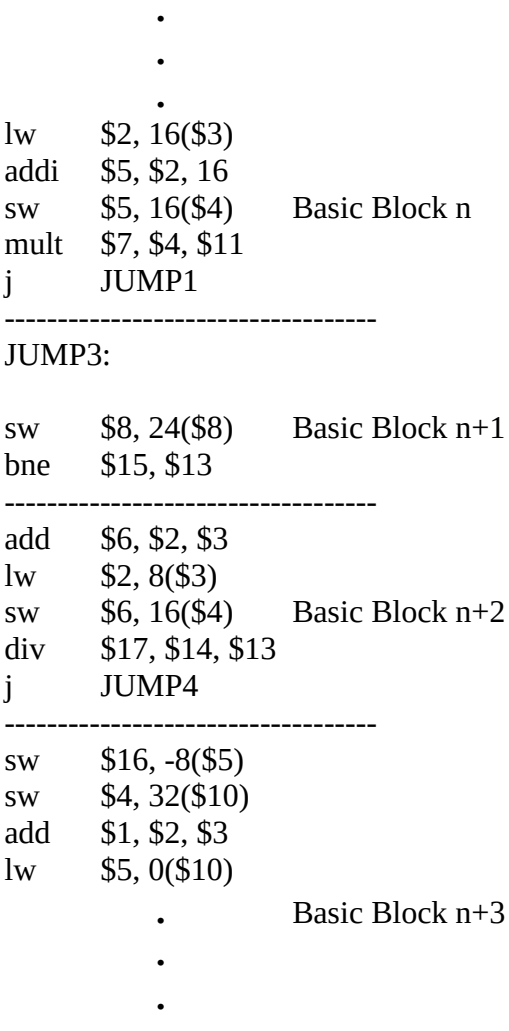

Figure 3.2. Sample Basic Block Segmentation of Source Code

level of abstraction, where the algorithm would attempt to work on "super-blocks" that contain blocks as it basic unit (as opposed to blocks that contain instructions as the basic unit). However, there no power characterization (a PDT) to use to calculate the self and successor forces of an entire block of code. In other words, it is not logical to make a generalization about what it means to categorize the cost of switching from one block to the next. Although developing an alternative algorithm to perform inter-block scheduling may be a cause for future work, it is not believed that significant power reduction will be achieved at the inter-block level. Correct code execution must be maintained and handling multiple branches to and from a basic block reduces its ability to be reordered.

3.5 Self and Successor Forces

Although FDS and FDIS are similar, the calculation of self and successor forces are somewhat different. This is primarily due to the domain unto which the algorithm is being applied. As mentioned previously, the algorithm uses the power dissipation table in the calculation of the self and successor forces. To do so, the algorithm first (in lines 7 through 12) finds any instruction that has a mobility (ALAP-ASAP) equal to zero and assigns that instruction to the one and only allowable time step. From there, the algorithm loops through each unscheduled time step, scheduling one of the unscheduled instructions to the unscheduled time frame. Therefore, when the last time step is scheduled, every time frame is assigned a particular instruction, from which a new schedule can be obtained. In order for the algorithm to select an instruction to schedule, it must first calculate the self and successor forces and find the minimum. Figure 3.3 shows how the self-force is calculated. Every unscheduled instruction is a possible instruction that can be placed in the current time step. Therefore, the algorithm must run for each unscheduled instruction before it can make a decision about which instruction has the minimum force. If ASAP=ALAP for any instruction, then the current time step is the last time step that the instruction can be scheduled into. Therefore, there is no need to calculate any forces. That instruction must be scheduled in the current time step. If this is not the case, the algorithm calculates the probabilities for each time step (see Equation 2.2). Once completed, the algorithm

tentatively assigns the current instruction to the current time step. From Equation 2.5, the self-force is equal to the sum of the forces at each time step. As previously mentioned, the spring constant in FDIS is the value in the power dissipation table (PDT). Calculating the force for the current time step is straightforward since it is assumed that the current instruction is being scheduled in that time slot and the instruction in the previous time step is already known since it was previously scheduled. Therefore, the power dissipation can easily be looked up in the PDT. However, the power dissipation values for the other time steps are not yet known since those time steps are to be scheduled after the current time step. For all time steps after the current time step that require a self force calculation, the self force is calculated using all of the unscheduled instructions as potential instructions to be scheduled just before the time step for which the self force is being calculated. The average of the self-forces is then taken and used as the self-force for that particular time step. Once the calculations of self-forces are completed, successor forces must be calculated as well. There are not predecessor calculations to be made in FDIS, since the time steps previous to the current time step are already scheduled and cannot be altered. Therefore, predecessor values in FDIS are always zero. Calculation of successor forces is quite similar to the self-force calculation and is shown in Figure 3.4. Successor forces are those forces incurred in moving an instruction to a different time slot. As a result, the first task is to ensure that all instructions still have mobility. If ASAP=ALAP, then the instruction cannot be moved and the successor force is infinite. There is no need to continue further. However, if all instructions do have a mobility, then the probabilities for each instruction can be found. To find the complete successor force, the sum of all of the instructions from their current time slots must be determined. Just as was the case in the self-force, the calculation of the successor force for the current time step is easy, since the power dissipation value can be looked up in the table. Even the successor force in the next time step can be determined, since it is assumed that the instruction whose successor force is being calculated will be scheduled in the current time step. However, similar to the self-force, any time steps beyond the current+1 time step must use the average successor force. Once the successor force is calculated, it can be added to the self-force, and a final scheduling decision for the current time step can be made.

```
For all unscheduled instructions
   tmp_ts = current time step
  If(ASAP == ALAP)
     Schedule Instruction to current time step
      Break
   Else
      Calculate Probabilities
   End If
   Assume Instruction assigned to current time step
   For(i = ASAP; i <= ALAP; i++)
      If(i == ASAP)
         index1 = PDT index  of inst. in previous time step
         index2 = PDT index value of current instruction
         x = Change in Probability
         self Force =+ x*PDT[index1][index2]
      Else
         index2 = PDT index value of current instruction
         counter = 0
         For all other unscheduled instructions j
               Temp = 0.0
               If(ASAP <= tmp_ts && ALAP >= tmp_ts)
                  index1 = PDT index  value of instruction j
                  x = Change in Probability
                   Temp =+ x*PDT[index1][index2]
                  counter++
               End If
         End For
         self_force = self_force + Temp / counter;
         tmp_ts++
      End If
   End For
End For
```

Figure 3.3. Calculation of Self-Forces

```
For all other instructions k
  tmp_ts = Current Time Step
  If(ASAP == ALAP)
    Instruction cannot be moved
    Break
  Else
    Calculate Probabilities
  End If
  Assume Instruction is moved
  for(i = ASAP; i <= ALAP; i++)
    If(i == ASAP)
      index1 = PDT index  of inst. in previous time step
      index2 = PDT index value of instruction k
      x = Change in Probability
      succ_force =+ x*PDT[index1][index2]
    Else If(i == ASAP + 1)
      index1 = PDT index  value of current instruction
      index2 = PDT index  value of instruction k
      x = Change in Probability
      succ_force =+  x* PDT[index1][index2]
    Else
      index2 = PDT index  value of instruction k
      counter = 0
      Temp = 0
      for all other instructions j
        If(ASAP <= tmp_ts && ALAP >= tmp_ts)
          index1 = PDT index  value of instruction j
            x = Change in Probability
          Temp =+  x*PDT[index1][index2];
          counter++;
        End If
      End For
      succ_force = succ_force + Temp/counter;
      tmp_ts++
    End If
  End For
End For
```

Figure 3.4. Calculation of Successor Forces

CHAPTER 4

EXPERIMENTAL RESULTS

4.1 Overview

To determine the actual power savings of the algorithm, testing was performed in four stages. In the first stage, all of the benchmarks were compiled using the SimpleScalar compiler, assembler, and loader in order to create a SimplePower executable. The resulting binary file is a baseline version of the original files, where no optimization has been performed. Therefore, this simulation provided the raw power consumption. The second stage entailed compiling the benchmarks using the SimplePower version of the SimpleScalar compiler, which included register renaming. This stage provided a baseline as to the power conservation that can be achieved with only register renaming, when compared to Run 1. The final two stages consisted of repeating the previous two stages, but first applying the new FDIS algorithm to the code. Figure 4.1 provides a flowchart of the different runs of the experiment performed. The results provide insight to the power savings of the algorithm. Repeating stage one with the FDS algorithm provides the raw power savings of the algorithm. Repeating the second stage reveals the true benefit of the algorithm. Even when code has been modified to incorporate register renaming, the FDIS algorithm still provides an additive power savings. This relationship validates the ability of the FDIS algorithm to be used in conjunction with other techniques to achieve maximum results.

4.2 Results

Tables 4.1, 4.2, and 4.3 summarize the power results achieved from Runs 1-4 (see Figure 4.1). For each benchmark, the original switch capacitance (in nF), the reduced

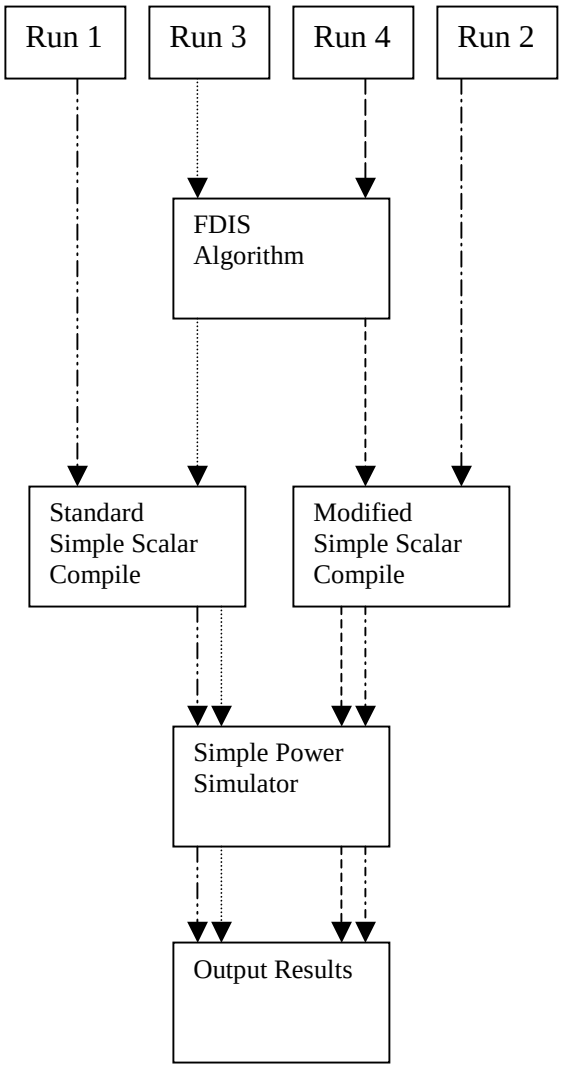

Figure 4.1. Flowchart of Experimental Runs

switch capacitance (in nF), and the percentage improvement is shown[1]. Table 4.4 reflects the performance improvement of the FDIS algorithm, with register renaming. Similar to the power tables, the Table 4.4 outlines the number of cycles each benchmark required before and after optimization, as well as the percentage reduction.

4.2.1 Power Results

From the tables, it can be seen that a significant power savings was seen in almost all of the benchmarks. The only exception was the binary search (bsrch.c) benchmark. Because of its small and simple structure, few opportunities within the code exist to perform optimization. However, this specialized benchmark demonstrates that in cases where few optimization decisions exist, the algorithm does not incorrectly select scheduling decisions that negatively affect performance or power.

Although the binary search benchmark only yielded a minute improvement, the quick sort benchmark (quick.c) yielded the maximum improvement. With power and performance improvements of over 24%, the benchmark had an order of magnitude gain over the other benchmarks. Much of this improvement can be explained due to the nature of the quick sort algorithm. The algorithm contains a simple code structure executed a number of times. From Table 4.1, register renaming alone can achieve the results seen from the FDIS algorithm, indicating that removal of stalls through the use of either register renaming or reordering of instructions to remove conflicts is the primary source of the improvement. As a result, all tables contain an average with and without the quick sort benchmark. This is done in order to remove the skew of the quick sort results on the overall average.

Tables 4.1 and 4.2 represent the overall independent performance of a register renaming scheme and the FDIS algorithm. Table 4.3 reflects the overall combined efforts of the two techniques. Although some of the benchmarks had slight decrease in the overall improvement, most of the benchmarks had a significant improvement. One especially positive result was the permutation benchmark (perm.c). Individually, the register renaming and FDIS techniques only introduced a 0.50% and 0.16% improvement, respectively. Com-

[1]Power consumption is a function of switch capacitance and voltage. Therefore, all comparisons or derivations made using switch capacitance will correlate to the actual power savings seen for a given system

Table 4.1. Switch Capacitance Reduction/Power Savings using Register Renaming

Benchmark	Original (nF)	Rescheduled (nF)	Reduction
bsrch.c	11500	11493	0.06%
bubble.c	12365628	11709060	5.31%
hanoi.c	6341659	6206131	2.14%
heap.c	3598277	3587242	0.31%
matmult.c	110309	108584	1.56%
perm.c	24001185	23880623	0.50%
quick.c	2077771	1578328	24.04%
Average w/ quick.c			4.85%
Average w/o quick.c			1.65%

Table 4.2. Switch Capacitance Reduction/Power Savings for FDIS

Benchmark	Original (nF)	Rescheduled (nF)	Reduction
bsrch.c	11500	11485	0.13%
bubble.c	12365628	11335292	8.33%
hanoi.c	6341659	6186881	2.44%
heap.c	3598277	3511171	2.42%
matmult.c	110309	107036	2.96%
perm.c	24001185	23961281	0.16%
quick.c	2077771	1522770	26.70%
Average w/ quick.c			6.17%
Average w/o quick.c			2.74%

Table 4.3. Switch Capacitance Reduction/Power Savings for FDIS with Register Renaming

Benchmark	Original (nF)	Rescheduled (nF)	Reduction
bsrch.c	11500	11491	0.08%
bubble.c	12365628	11303461	8.58%
hanoi.c	6341659	6170045	2.70%
heap.c	3598277	3522002	2.11%
matmult.c	110309	106823	3.16%
perm.c	24001185	23280466	3.00%
quick.c	2077771	1566066	24.62%
Average w/ quick.c			6.33%
Average w/o quick.c			3.27%

bined, however, resulted in a 3.00% improvement. Overall, the register renaming scheme increased the improvement of the FDIS algorithm over 0.50%. Although this value does not appear to be significant, 0.50% is nearly a 20% improvement over the FDIS only version.

4.2.2 Performance Results

Table 4.4 represents the overall clock cycle reduction of the combined FDIS and register renaming scheme. Comparing the average results of the cycle reduction (2.76%) to the average power reduction seen in Table 4.3 (3.27%), the two values are relatively close. Furthermore, an analysis of the two tables demonstrates a correlation between reduced number of clock cycles and reduced power consumption. This phenomenon reflects that much of the power savings attained in Tables 4.1, 4.2, and 4.3 are a result of reduced clock cycles. Although some power improvement was gained through the minimization of the inter-instruction cost, like in case of the binary search algorithm, where there was a small power reduction despite the lack of a performance gain, virtually all of the savings must be attributed to clock cycle reduction. This correlation does not affect the net result of the algorithm, but such results do explain the intrinsic properties of the FDIS technique.

In subsection 3.2.1, operand characterization was discussed. The idea behind creating an operand power table was to consider the actual operands in calculating the overall power cost of instruction, rather than just the operation itself. Experimentation performed in this area proved futile, as no characterization created a positive effect on the overall power or performance of the benchmarks. The realization that the cycle reduction accounts for

Table 4.4. Clock Cycle Reduction/Performance Improvement

Benchmark	Original(Cycles)	Rescheduled(Cycles)	Improvement
bsrch.c	390	390	0.00%
bubble.c	391398	364899	6.77%
hanoi.c	220820	217772	1.38%
heap.c	105380	103159	2.10%
matmult.c	4005	3861	3.59%
perm.c	751789	731626	2.68%
quick.c	67543	51341	23.98%
Average w/ quick.c			5.79%
Average w/o quick.c			2.76%

virtually all of the gains seen validate this result. Minimizing the switching activity on the address bus will not reduce the number of cycles and therefore, will not translate into a power or performance gain.

Another interesting side note to the observed results is the ability of the algorithm to optimize for power, but actually schedule for performance. This intrinsic trait is especially beneficial in today's processing environment. Using power as a scheduling metric to yield a performance improved schedule ensures that the schedule attained reduces power while improving performance. Such approaches may very well be the paradigm upon which new algorithms must use.

CHAPTER 5

CONCLUSIONS AND FUTURE RESEARCH

Providing increasing performance, while reducing the power requirements to reach the desired performance goals, is a critical factor facing the computing industry. Future solutions will likely involve the use of many techniques, at all levels of abstraction. As a result, newly developed techniques should not only produce significant results, but should also be combinable with other techniques, in order to achieve maximum performance while using minimal power.

In this paper, we presented a new instruction scheduling technique based on classical force directed scheduling. The technique attempts to reduce power and in the process, also attempts to improve performance. The technique is combinable with other techniques, such as register renaming. Through the use of this algorithm, in combination with register renaming, an average power reduction of 3.25% was obtained with a reduction of up to 26% seen for selected benchmarks. In addition, a performance gain of 2.8% with a maximum of 24% was attained.

In order to more fully mature the algorithm and it capability, many areas of research remain. The running time of the FDIS algorithm is $O(n^3)$. Ideally, the running time should be an order of magnitude less. Therefore, future research should concentrate on running time reduction. Additionally, a more thorough analysis should be performed on inter-block scheduling. Currently, no information is available to determine how close the FDIS schedule is to the globally optimal solution. This determination should further validate the results seen in this work. A more thorough analysis should also be conducted in order to determine if any other intrinsic properties could be exploited by FDIS. Finally, more experimentation should be performed on a wider range of benchmarks, including those that are larger and contain more instruction types, such as floating type instructions.

REFERENCES

[1] N. Vijaykrishnan A. Parikh, M. Kandemir and M.J. Irwin. Instruction Scheduling based on Energy and Performance Constraints. In *Proceedings. IEEE Computer Society Workshop on VLSI*, pages 37–42, Orlando, Florida, April 2000.

[2] N. Vijaykrishnan A. Parikh, M. Kandemir and M.J. Irwin. VLIW Scheduling for Energy and Performance. In *Proceedings. IEEE Computer Society Workshop on VLSI*, pages 111–117, Orlando, FL, April 2001.

[3] J. Lee C. Lee and T. Hwang. Compiler Optimization on Instruction Scheduling for Low Power. In *Proceeding. The 13th International Symposium on System Synthesis*, pages 55–60, Madrid, Spain, September 2000.

[4] V. Tiwari D. Brooks and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings. International Symposium on Computer Architecture*, pages 83–94, Vancover, Canada, June 2000.

[5] D. Dobberpuhl. The design of a high-performance low-power microprocessor. In *Proceedings. Int. Symp. Low Power Electronics and Design*, pages 11–16, Monterey, CA, August 1996.

[6] P. Dongale. *Force-Directed Instruction Scheduling for Low Power* . University of South Florida. Department of Computer Science and Engineering, 2003.

[7] M. Kandemir G. Esakkimuthu, N. Vijaykrishnan and M.J Irwin. Memory System Energy: Influence of Hardware-Software Optimizations. In *Proceedings. The 2000 International Symposium on Low Powere Electronics and Design*, pages 244–246, Rapallo, Italy, July 2000.

[8] S. Gupta and S. Katkoori. Force-directed scheduling for dynamic power optimization. In *Proceedings. IEEE Computer Society Annual Symposium on VLSI*, pages 68–73, Pittsburg, PA, April 2002.

[9] M. Gupta J. Kin and W. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proceedings. IEEE Symp. Microarchitecture*, pages 184–193, Research Triangle Park, NC, Dec 1997.

[10] M. Irwin W. Ye M. Kandemir, N. Vijaykrishnan. Influence of Compiler Optimizations on System Power. *IEEE Transactions on VLSI Systems*, 9:801–804, 2001.

[11] L. John M. Valluri and H. Hanson. Exploiting Compiler-Generated Schedules for Energy Savings in High-Performance Processors. In *ISLPED'03*, pages 414–419, Seoul, Korea, August 2003.

[12] V. Tiwari S. Malik and A. Wolfe. Compilation Techniques for Low Energy: An Overview. In *Proceedings Design Automation Conference*, pages 38–39, San Diego, CA, October 1994.

[13] N. Bellas, et al. Architectural and Compiler Techniques for Energy Reduction in High-Performance Microprocessors. *IEEE Transactions on VLSI Systems*, 8:317–326, 2000.

[14] P. Ong and R. Yan. Power-Conscious Software Design - a framework for modeling software on hardware. In *Digest of Technical Papers. IEEE Symposium on Low Power Electronics*, pages 36–37, San Diego, CA, October 1994.

[15] P.G. Paulin and J.P. Knight. Force-directed scheduling in automatic data path synthesis. In *Proceedings. 24th Design Automation Conference*, pages 195–202, Miami Beach, FL, July 1987.

[16] P.G. Paulin and J.P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE Transactions on Computer-Aided Desing of Integrated Circuits and Systems*, 8:661–679, 1989.

[17] H. Kojima et. al R. Bajwa, M. Hiraki. Instruction buffering to reduce power in processors for signal processing. *IEEE Transactions on VLSI Systems*, 5:417–424, 1997.

[18] M.J. Irwin R. Chen R. Mehta, R.M Owens and D. Ghosh. Techniques for low energy software. In *Proceedings. International Symposium on Low Power Electronics and Design*, pages 72–75, Monterey, CA, August 1997.

[19] D. Carmean S. Gunther, F. Binns and J. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, pages 1–17, 2001.

[20] D. Grunwald S. Manne and A. Klauser. Piepeline gating: Speculation control for energy reduction. In *Proceedings. Int. Symp. Computer Architecture*, pages 132–141, Barcelona, Spain, June 1998.

[21] E. Larson T. Austin and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35:59–67, 2002.

[22] M. Kandemir W. Ye, N. Vijaykrishnan and M.J. Irwin. The Design and Sse of SimplePower: A Cycle-Accurate Energy Estimation Tool. In *Proceedings Design Automation Conference*, pages 340–345, Los Angeles, CA, June 2000.