Energy Efficient Branch Prediction

Michael Andrew Hicks

A thesis submitted in partial fulfilment of the requirements of the
University of Hertfordshire for the degree of Doctor of Philosophy

December 2007

To my family and friends.

Contents

1 Introduction **1**
- 1.1 Thesis Statement . . . 1
- 1.2 Motivation and Energy Efficiency . . . 1
- 1.3 Branch Prediction . . . 3
- 1.4 Contributions . . . 4
- 1.5 Dissertation Structure . . . 5

2 Energy Efficiency in Modern Processor Design **7**
- 2.1 Transistor Level Power Dissipation . . . 7
  - 2.1.1 Static Dissipation . . . 8
  - 2.1.2 Dynamic Dissipation . . . 9
  - 2.1.3 Energy Efficiency Metrics . . . 9
- 2.2 Transistor Level Energy Efficiency Techniques . . . 10
  - 2.2.1 Clock Gating and V_{dd} Gating . . . 10
  - 2.2.2 Technology Scaling . . . 11
  - 2.2.3 Voltage Scaling . . . 11
  - 2.2.4 Logic Optimisation . . . 11
- 2.3 Architecture & Software Level Efficiency Techniques . . . 11
  - 2.3.1 Activity Factor Reduction . . . 12
  - 2.3.2 Delay Reduction . . . 12
  - 2.3.3 Low Power Scheduling . . . 12
  - 2.3.4 Frequency Scaling . . . 13
- 2.4 Branch Prediction . . . 13
  - 2.4.1 The Branch Problem . . . 13
  - 2.4.2 Dynamic and Static Prediction . . . 14
  - 2.4.3 Dynamic Predictors . . . 15
  - 2.4.4 Power Consumption . . . 18
- 2.5 Summary . . . 18

3 Related Techniques **20**
- 3.1 The Prediction Probe Detector (Hardware) . . . 20
  - 3.1.1 Implementation . . . 20
  - 3.1.2 Pipeline Gating . . . 22

3.2 Software Based Approaches . . . 23
3.2.1 Hinting and Hint Instructions . . . 23
3.3 Analysis and Summary . . . 24

4 Initial Investigation and Preliminary Research 26
4.1 Research Question Focus . . . 26
4.2 Static Methods to Avoid Dynamic Branch Prediction . . . 27
4.2.1 Delay Region Scheduling . . . 27
4.2.2 Static Prediction and Instruction Hints . . . 29
4.2.3 Guarded Execution . . . 30
4.3 Hardware Multithreading . . . 31
4.4 Initial Experiments . . . 31
4.4.1 Removing Dynamic Branch Predictors . . . 31
4.4.2 Instruction Stream Research (HTracer) . . . 33
4.4.3 I-Cache Experimentation . . . 34
4.5 Summary . . . 34

5 The Combined Approach 36
5.1 Local Delay Region Scheduling . . . 36
5.2 Profiling . . . 38
5.2.1 Assigning a Static Branch Behaviour . . . 39
5.2.2 Adaptive Branch Bias Measurement (ABBM) . . . 40
5.3 The Combined Algorithm . . . 40
5.4 Hardware Implementation . . . 41
5.4.1 Instruction Set Modifications . . . 42
5.4.2 Hardware Modifications . . . 44
5.5 Summary . . . 46

6 Simulation Tools 47
6.1 Introduction . . . 47
6.2 Simulator (HWattch) . . . 47
6.2.1 Architecture Model . . . 49
6.2.2 Architecture Modifications . . . 52
6.2.3 Profiling Enhancement . . . 55
6.2.4 Instruction Set (PISA) . . . 56
6.2.5 Compiler (Custom GCC) . . . 59
6.3 Scheduler and Static Prediction Assigner (HACA) . . . 59
6.3.1 Combined Algorithm: Practical Implementation . . . 59
6.4 EEMBC . . . 63
6.4.1 Sub-Suites and Benchmarks . . . 63
6.4.2 Bespoke Build System for the Combined Algorithm . . . 65
6.5 Summary . . . 65

7 Simulations and Results **66**
7.1 Introduction . . . 66
7.2 The Baseline Models . . . 66
7.2.1 The Branch Predictor . . . 67
7.2.2 Scalar Processor . . . 67
7.2.3 Multiple Instruction Issue Processor . . . 68
7.3 Preamble To Results . . . 70
7.3.1 Metrics . . . 70
7.3.2 Calculation of Averages and 'Weighted Averages' . . . 74
7.3.3 Important Summary Notes . . . 75
7.4 Scalar Processor Results . . . 75
7.4.1 Benchmark Breakdown . . . 75
7.4.2 Averages . . . 78
7.5 Two Instruction Issue Processor Results . . . 79
7.5.1 Benchmark Breakdown . . . 79
7.5.2 Averages . . . 79
7.6 Sixteen Instruction Issue Processor Results . . . 83
7.6.1 Benchmark Breakdown . . . 83
7.6.2 Averages . . . 86
7.7 Overall Analysis . . . 86
7.7.1 Results Summary . . . 88

8 Comparisons and Enhancements **90**
8.1 Comparison of ABBM with Fixed Bias Level and Compiler Heuristics . . . 90
8.1.1 Results and Analysis . . . 91
8.2 Reducing Set Associativity in the Branch Target Buffer . . . 92
8.2.1 Results and Analysis . . . 92
8.3 Summary . . . 94

9 Conclusion and Discussion **96**
9.1 Thesis Summary . . . 96
9.1.1 Key Novelties and Contributions . . . 97
9.2 Generalisation . . . 97
9.3 Critique . . . 99
9.3.1 Local Delay Region . . . 99
9.3.2 Hint Bits . . . 100
9.3.3 Timing Issues . . . 101
9.3.4 Profiling Duration . . . 102
9.3.5 Profiling on a 'Real' Architecture . . . 102
9.3.6 Dependency on Datasets . . . 104
9.4 Related Work Comparison . . . 104
9.4.1 Prediction Probe Detector . . . 105
9.5 Future Work . . . 106

9.5.1 Maximising the Fetch Window of Wide Issue Processors . 106
9.5.2 Hinting Libraries 107
9.5.3 Combining with the Prediction Probe Detector 107
9.5.4 Hints and Context Switching 108
9.5.5 Profiling and Processor-Wide Power Saving 109
9.6 Concluding Remarks 109

Bibliography **111**

Glossary **120**

Appendix A: Published Papers
i Towards an Energy Efficient Branch Prediction Scheme...
ii Reducing the Branch Power Cost In Embedded Processors...
iii HTracer: A Dynamic Instruction Stream Research Tool
iv Enhancing the I-cache to Reduce the Power Consumption...

Appendix B: Technical Reports
i An Introduction to Power Consumption Issues in Processor Design
ii HTracer V0.5: A User Guide

Appendix C: Additional Background

Appendix D: Raw Data

List of Figures

2.1 An example five stage processor pipeline 14
2.2 An example of a modern dynamic predictor architecture 15

3.1 The Prediction Probe Detector 21

5.1 An example of local delayed branch scheduling 37
5.2 The basic structure of profiling 39
5.3 Block model of the profiling and hinting regime 42
5.4 Hardware modifications required in the instruction fetch stage . . 45

6.1 The Wattch simulator in relation to SimpleScalar 48
6.2 The Wattch simulator pipeline 49
6.3 A logical represention of the IF stage hint-bits showing '1,1' . . . 53
6.4 A logical represention of the EXE stage hint-bits showing '1,0' . . 54
6.5 The PISA instruction format 56
6.6 The location of the two hint-bits within the branch instruction format 58

7.1 Scalar baseline global power savings (%) compared with ideal (free) prediction 77
7.2 Scalar baseline average global power savings (%) compared with ideal (free) prediction 78
7.3 2-way issue baseline global power savings (%) compared with ideal (free) prediction 81
7.4 2-way issue baseline average power savings (%) compared with ideal (free) prediction 82
7.5 16-way issue baseline global power savings (%) compared with ideal (free) prediction 85
7.6 16-way issue baseline average power savings compared with ideal (free) prediction 87

8.1 Average Change in the dynamic instruction stream after resizing the BTB from four-way to two-way set-associativity 93
8.2 Additional power saving after resizing the BTB 94

9.1 Series and parallel i-cache/branch predictor access. (1) and (2) represent the direction and target address predictors, respectively . 101

List of Tables

5.1 Static and dynamic branch occurrence for each PISA branch, and its occurrence across the whole EEMBC benchmark suite 43

6.1 Static and dynamic branch occurrence for each PISA branch, and its occurrence across the whole EEMBC benchmark suite 57
6.2 The full EEMBC benchmark suite with descriptions 64

7.1 Scalar Processor Baseline Configuration 69
7.2 2-Way Issue Processor Baseline Configuration 71
7.3 16-Way Issue Processor Baseline Configuration 72
7.4 Benchmark breakdown results for scalar baseline processor 76
7.5 Average benchmark results for scalar baseline processor 78
7.6 Benchmark breakdown results for two-way issue baseline processor 80
7.7 Average benchmark results for two-way issue baseline processor . 82
7.8 Benchmark breakdown results for sixteen-way issue baseline processor 84
7.9 Average benchmark results for sixteen-way issue baseline processor 86

8.1 Average benchmark results for fixed bias hinting 91

Acknowledgements

I would first like to thank Colin Egan. His continual support, both academic and personal, played an essential role in this work. I would also like to thank Bruce Christianson for his insightful assistance and knowledge of the research process. The work described in this dissertation was also greatly aided by rational discussion with Patrick Quick – thankyou.

I am indebted to my parents and family. This work would not have been possible without them.

Finally, I would like to thank all of those remaining people that have, either directly or indirectly, assisted me in the completion of this project.

Abstract

Energy efficiency is of the utmost importance in modern high-performance embedded processor design. As the number of transistors on a chip continues to increase each year, and processor logic becomes ever more complex, the dynamic switching power cost of running such processors increases. The continual progression in fabrication processes brings a reduction in the feature size of the transistor structures on chips with each new technology generation. This reduction in size increases the significance of leakage power (a constant drain that is proportional to the number of transistors). Particularly in embedded devices, the proportion of an electronic product's power budget accounted for by the CPU is significant (often as much as 50%).

Dynamic branch prediction is a hardware mechanism used to forecast the direction, and target address, of branch instructions. This is essential to high performance pipelined and superscalar processors, where the direction and target of branches is not computed until several stages into the pipeline. Accurate branch prediction also acts to increase energy efficiency by reducing the amount of time spent executing mis-speculated instructions. 'Stalling' is no longer a sensible option when the significance of static power dissipation is considered. Dynamic branch prediction logic typically accounts for over 10% of a processor's global power dissipation, making it an obvious target for energy optimisation.

Previous approaches at increasing the energy efficiency of dynamic branch prediction logic has focused on either fully dynamic or fully static techniques. Dynamic techniques include the introduction of a new cache-like structure that can decide whether branch prediction logic should be accessed for a given branch, and static techniques tend to focus on scheduling around branch instructions so that a prediction is not needed (or the branch is removed completely).

This dissertation explores a method of combining static techniques and profiling information with simple hardware support in order to reduce the number of accesses made to a branch predictor. The local delay region is used on unconditional absolute branches to avoid prediction, and, for most other branches, Adaptive Branch Bias Measurement (through profiling) is used to assign a static prediction that is as accurate as a dynamic prediction for that branch. This information is represented as two hint-bits in branch instructions, and then interpreted by simple hardware logic that bypasses both the lookup and update phases for appropriate branches.

The global processor power saving that can be achieved by this Combined Algorithm is around 6% on the experimental architectures shown. These architectures are based upon real contemporary embedded architecture specifications.

The introduction of the Combined Algorithm also significantly reduces the execution time of programs on Multiple Instruction Issue processors. This is attributed to the increase achieved in global prediction accuracy.

Chapter 1

Introduction

This dissertation investigates one of the most important areas of contemporary research in computer engineering: energy efficiency [117] [68]. With evermore high performance embedded devices appearing on the market each year [71], requiring more and more power from a fixed capacity battery, the need to improve the energy efficiency of processor designs is paramount to the viability of new architectures [13]. This project proposes a thesis that the energy efficiency of the costly branch predictor unit can be further increased by a combination of traditional and new techniques [44].

This chapter introduces the motivation and scope of this project. The broad thesis and contributions are described, and the structure of the rest of dissertation is shown.

1.1 Thesis Statement

The resulting thesis of this PhD project is that software and hardware prediction techniques, traditionally used exclusively, can be combined to reduce the power consumption of dynamic branch prediction units in modern processors. This is achieved with minimal hardware modification and shows that hardware and software prediction strategies can be further combined to increase global processor energy efficiency.

1.2 Motivation and Energy Efficiency

> We're projecting by 2010 there will be more than 2.5 billion wireless handheld devices capable of providing the communications functions combined with the processing power of today's high-performance PCs.

– Paul Otellini, Intel, 2003 [75]

Energy efficiency is an increasingly important issue in the field of computer engineering. This can be said because power consumption affects so many factors in real world implementations:

Battery Life - The most obvious restriction on mobile devices is the length of time that the device can be used for and the performance that can be expected. Power efficient designs can extend this but, as shown later, careful consideration must be taken in assessing the relative advantages of a design.

Thermal Issues - Power dissipation results in heat. Excessive heat dissipation will affect the design of relevant cooling systems (and thus device packaging/size), reliability and precise timing. One of the most important issues on the desktop machine is packaging size and cost. On large scale servers packaging size will represent a volumetric limit on rack capacity.

Large Scale Power Consumption - This aspect often seems irrelevant when examining the desktop and small scale market, but when one looks at massive arrays of machines the resulting power consumption can be substantial. Small power savings over a large number of processors account for a significant saving for a single user/organisation.

Given the prediction quoted by Intel, one can see the extreme importance of power efficient designs in the coming decades. In fact, power efficiency is a likely lynchpin for new high speed processor designs. If new portable devices are to achieve the prediction of Intel then they must be both energy efficient and high performance [38]. Such requirements are pulling processor design in two directions at once; high performance and low power was often considered to be a dichotomy.

Processors consume power at the transistor level by both switching and maintaining their state. When a transistor switches it leaks a small amount of electricity into the circuit. When transistors are maintaining their state a small amount of electricity can escape through the gate due to imperfections in the production process. Transistors are caused to switch when activity occurs in the processor. This includes accesses to caches, functional units and the algorithms used to manage processor activity. With processors now containing over one billion transistors, the issues of power consumption and energy efficiency are more important than ever, particularly to the embedded market.

The thesis of this document is that it is possible to maintain the high performance of a processor, through the use of branch prediction, but relieve some of the significant power burden of high performance designs. The central focus is the embedded market as this is where energy efficiency is of the utmost importance. To date, most research into energy efficiency has followed the traditional separation of transistor level, dynamic and static approaches. There has been little in the way of handshaking between static and dynamic techniques. The result of the work

conducted in this report demonstrates how static and dynamic techniques can be combined into a more effective energy scheme.

If embedded processors continue to increase in performance, there *must* be a coordinated effort to develop energy efficient algorithms [69]. If this does not happen, the new generation of embedded devices will have an insatiable rate of power consumption for current battery technology.

1.3 Branch Prediction

Modern high performance embedded processors use a technique known as pipelining to increase instruction level parallelism. Pipelining breaks down the processor into a series of different stages with an intermediate store between each stage [84]. This allows a number of instruction to be undergoing execution at any given time. This increases the throughput of the processor, in instructions per second, because a higher number of instruction complete during each clock cycle (as opposed to a single instruction using the entire processor for several cycles). Increasingly common is that a processor can fetch and issue several instructions in a given clock cycle. This is referred to as Multiple Instruction Issue (MII) and increase instruction level parallelism still further [84].

The process of pipelining means that each instruction requires several stages of 'execution' before the result of the given instruction is known. This means that there are several instructions undergoing execution in the preceding pipeline stages. When fetching instructions, this delay before the result of execution poses a particular problem for one class of instructions – branches. A branch instruction changes the flow of instructions, and hence the fetch unit in the processor must know which part to fetch instructions from in order to fully utilise all of the pipeline stages behind the branch instruction. However, the correct path to fetch from is not known until the branch resolution pipeline stage. Filling these pipeline slots is paramount to the performance of the processor when it is considered that around one in every eight instructions is a branch [31].

The technique generally used to overcome this performance problem is called branch prediction [31]. Branch prediction attempts to predict which way a branch instruction will go before the actual behaviour is known. Dynamic Branch predictors, which use dedicated hardware, can be highly accurate (as high as 98% correct), but do still inevitably make mispredictions. These mispredictions require recovery logic to stop the pipeline and flush the incorrectly fetched instructions. Such mispredictions pose a severe performance penalty and should be avoided at all costs [93].

Branch prediction has traditionally taken place statically or dynamically. Static prediction takes place at compile time of a program and can involve a variety of techniques such as hint-bits (which give the processor an idea of which way the branch might go) and delay region scheduling (a decision is made about which way

the branch might go, and relevant instructions are used to fill the stages behind the branch). Dynamic prediction uses a special unit in the processor hardware which is updated each time a branch is encountered with information about how it behaved. This prediction unit is then used by the instruction fetch logic to decide which way a branch is likely to go. Dynamic branch predictors are largely the favoured method for branch prediction as they are the most accurate and simple to use. A fuller discussion of dynamic branch prediction and branch predictor types is given in the next chapter (2.4).

A dynamic branch predictor accounts for a significant portion of a processor's global power budget (around 10%), but it's accuracy is integral to the energy efficiency of the processor as a whole [82].

This thesis is the result of an investigation where the global processor power consumption was reduced by increasing the energy efficiency of the dynamic branch predictor, but without negatively impacting on its accuracy. This was achieved through the realisation that not all branches require a dynamic prediction, and that, with appropriate logic, the number of accesses made to a dynamic branch predictor can be significantly reduced by a combination of profiling and static scheduling techniques [44].

1.4 Contributions

This thesis makes contributions in both the fields of branch prediction and energy efficiency. All experimentation is conducted using the EEMBC embedded benchmarks and a variant of the Wattch processor power simulator.

The experimental work contained within this document demonstrates that the power consumption of a dynamic branch predictor unit can be reduced by lowering the number of accesses made to it by the processor logic at run time. This is achieved by combining several existing techniques, and some new, to form a modified branch prediction scheme.

Traditionally, branch prediction has occurred as either a static or dynamic process [33]. Either there was no dynamic branch predictor, and the compiler assigned static branch predictions, or the dynamic predictor negated the need to static prediction. Some architectures, such as the PowerPC, allow for assigning hints in certain heavily biased branch instructions to override a dynamic prediction in the hope of increasing accuracy.

A key proposal of this thesis is the use of an Adaptive Branch Bias Measurement (ABBM). The combined algorithm uses profiling data about the behaviour of branches and the dynamic branch predictor to permit the use of static hints in an instruction for only those instructions that do not require a dynamic prediction. This is combined with static scheduling, for certain branches, in order to significantly reduce the number of accesses made to the dynamic predictor unit. Simple modification of hardware logic permits the avoidance of accessing the predictor

and advanced calculation of the target address of certain branches. As a result of the application of the algorithm discussed in this report, a significant global power saving can be achieved.

The proposed algorithm is also shown to increase performance (by increasing branch predictor accuracy) and so is of interest to the branch prediction community in general.

The significance of this thesis lies in its elegance and ease of implementation using existing hardware and software. Large gains in energy efficiency can be achieved with minimum modifications in the design hierarchy.

1.5 Dissertation Structure

This dissertation is broken down into chapters. Each chapter builds on the last and leads into the next. Individual chapters are not intended to be entirely independent, and should be read with respect to their context in the structure. The chapters deal with the following areas:

1. Introduction.
2. Energy Efficiency in Modern Processor Design – This chapter provides a review of energy efficiency through all levels of abstraction in the design process. This includes an explanation of how power is dissipated in electronic circuits from an elementary level. Finally the chapter concludes with an introduction to dynamic branch predictors and how they consume power.
3. Energy Efficient Branch Prediction (related work)
4. Preliminary Research – This chapter investigates some of the questions which need to be answered before a viable energy efficient branch prediction scheme can be proposed. Existing static scheduling and prediction methods are examined, along with hardware multithreading. The results of some initial experiments are presented.
5. The Combined Approach – This chapter proposes the energy efficient branch prediction scheme for experimentation. The algorithm combined some traditional techniques with a new metric and hardware modifications. The specification of the algorithm here is intentionally abstract for discussion purposes.
6. Simulation – This chapter describes and discusses the experimental method, tools and architecture used to conduct experimentation with the combined algorithm in the next chapter. This chapter also provides a more concrete, implementation based, description of how the combined algorithm is implemented in hardware and software.

7. Experimentation and Results – This chapter uses the simulation tools and methods discussed in the previous chapter to evaluate the proposed combined algorithm on a number of baseline processor models. These models are discussed and the results are presented with a description of the evaluation metrics chosen.

8. Comparisons and Enhancements – This chapter compares the combined algorithm with some other techniques and examines how it could be extended by modifying the hardware of a processor to achieve further power savings. The applicability to a real architecture is then briefly discussed by comparison to the PowerPC instruction set.

9. Conclusion and Discussion – Finally, the overall novelties and contributions are stated in condensed form. This is followed by a discussion of any criticisms of the combined algorithm and the experimentation method. The effectiveness of the combined algorithm is compared to the most closely related work. The opportunities for future work are then presented. Finally, following this, are the concluding remarks of the thesis.

Chapter 2

Energy Efficiency in Modern Processor Design

The task of constructing energy efficient processors has been extensively examined at almost all levels of abstraction [115]. This chapter offers a basic introduction to the important low-level causes of energy loss in modern processors and explains the main areas of research in processor energy efficiency. Additional background, from an elementary level, can be found in Appendix B and Appendix C.

2.1 Transistor Level Power Dissipation

All CMOS circuits require a power source, and do not function without using some energy/power from this source. There are several ways in which this power is dissipated in CMOS circuits; the key areas of power dissipation are discussed in this section. While there is much research into reducing the loss of power in CMOS circuits [22] [27] [15], it must be noted that this power dissipation can never be reduced to zero as a current always needs to flow.

Equation 2.1 is the accepted [46] approximation for power dissipation in modern CMOS circuits.

$$Power \sim \frac{1}{2} C V^2 f A \tag{2.1}$$

Each term in the equation is defined as follows:

C – The Capacitance of the circuit. This is a function of all of the transistor interconnects which must be charged and discharged during circuit activity.

V – The Voltage that the circuit is running at. It is important to note that the Voltage is found to have a quadratic effect on power dissipation.

f – The clock Frequency of the circuit/processor.

A – The average Activity Factor of the circuit. That is, how often do transistors/gates, on average, make the transition from $1 \rightarrow 0$ and $0 \rightarrow 1$. A number between zero and one.

The power lost according to this general rule can be divided into two main varieties (and then subdivided further): static and dynamic power dissipation. These are explained in the following two sections.

2.1.1 Static Dissipation

Static power dissipation, or leakage, refers to a constant 'per-cycle' energy drain on the on the power source by the circuit. Traditionally, CMOS circuits have been classed as having almost no power leakage. However, this is no longer true [73]. Consequently, anything that increases the delay of a circuit (stalling in branch prediction, for instance) is now inefficient in terms of energy.

Sub-Threshold Leakage

For a transistor to function as a switch it has a property called the 'Switching Threshold'. This is the voltage on the gate terminal at which the transistor will permit a current to flow between the source and drain (or base and emitter in some nomenclature) – see Appendix C. The threshold is generally set by the chemical properties of the semiconductor material used by the circuit designers, but can be controlled dynamically in some circuits [6].

Equation 2.1 shows that voltage has a quadratic effect on the power dissipation of a circuit/processor. As such, processor designers have been reducing the core voltage at which processors run at in order to save power [27]. This means that all of the transistors in the control logic of the processor must be built with a lower threshold in order to maintain the correct behaviour. The result of this is that the difference between the drain voltage (i.e. 0 Volts) and the threshold of the transistor becomes much smaller. When the gap between the 'off' and 'on' voltage that a CMOS transistor is designed to work with becomes very close, the transistor will begin to allow small amounts of current to flow even when it is supposed to be switched off. These currents are known as Sub-Threshold Leakage currents, and are now the leading cause of static power dissipation in CMOS circuits [73].

As core voltages have been progressively scaled down to save power, sub-threshold leakage has increased to the extent that it can now account for up to 50% of all CMOS processor power loss [73]. The amount of power lost to sub-threshold leakage is proportional to the number of transistors in the circuit/processor.

2.1.2 Dynamic Dissipation

Dynamic dissipation is still generally the leading cause of power dissipation in most CMOS circuits, and is entirely proportional to the behaviour of the circuit. As such, it is highly interesting to hardware designers at all levels of abstraction.

Capacitive

Capacitive power is dissipated when a transistor changes state from high to lower or vice-versa. It is proportional to the energy that is stored on all of the interconnects between two state-changing transistors. During a state transition, the interconnects from the output of the changing transistor must be either charged or discharged in order to change to the correct output level. This process necessarily consumes power from the source. The factors directly affecting this dissipation are: interconnect thickness, circuit layout design, voltage and circuit activity [22].

Short Circuit

When a coupling of CMOS transistors change state, such as those shown in Appendix C, it does not happen instantly. There is a propagation delay while the input interconnects are charged/discharged to the required voltage. During this transition time, there is a brief point at which both transistors in a CMOS coupling are conducting (since they are both 'mid-threshold'). This allows a short circuit current to flow directly between the source and drain, and dissipates power. The factors directly affecting this dissipation are: transistor design properties, voltage and circuit activity [22].

2.1.3 Energy Efficiency Metrics

In order to properly quantify and compare the energy efficiency of different circuit designs, the following metrics are often used:

$Power \times time$ **(Energy Per Operation)** – A measure of the Power used during the course of a particular operation. This is equal to the Average Power used multiplied by the duration of the operation. In effect, this is a measure of how much electrical energy was used performing a particular operation.

Et^2 – Energy multiplied by time squared (also rewriteable as Pt^3) [68]. The Et^2 metric was developed as a comparison metric for cross-processor analysis since it is designed to be voltage independent. It is proportional to $\frac{MIPS^3}{Power}$. However, by eliminating dependence on voltage, this metric gives a heavy bias to delay/execution time.

A more detailed discussion of these and other metrics is available in Appendix A.

2.2 Transistor Level Energy Efficiency Techniques

Although the focus of this dissertation will be on a higher level of energy abstraction, there are several transistor level efficiency techniques which should be explained. The behaviour of these techniques can be affected by higher level decisions.

2.2.1 Clock Gating and V_{dd} Gating

'Gating' refers to the process of using a logic gate to switch off something in a circuit. In the case of energy efficiency, this usually refers to two processes: clock gating and V_{dd} gating [73].

Clock Gating

The clock signal of a synchronous circuit has a direct effect on the power dissipation in that circuit. This occurs as a result of the switching activity associated with that signal. As clock rates have increased, the problem of clock signal power dissipation has become important. In response to this, clock gating was introduced. When different units of a processor are found to be idle, the control logic of the processor will switch off the clock signal for that particular unit. This avoids dissipating power for the unit's period of inactivity. Clock gating stops a unit from functioning, but does not cause its state to be lost [73].

V_{dd} Gating

Additionally, when a unit is found to be idle, V_{dd} gating can also be used. When the on-chip control logic has found a unit to be idle, the entire drain connection to that part of the chip can be gated. This has the effect of actually powering down that unit of the chip. The advantage of this is a period of near zero power dissipation. However, V_{dd} gating can only be carried out when the unit is not required to store a particular state [73].

Defining Idle

When a unit should be marked as idle, and thus use a gating procedure, is decided by complex on-chip control logic. The control logic must monitor the activity of particular units and also use information about what kind of instructions are in the pipeline. Crucially, the behaviour of the program can decide how effective a gating regime will be; long bursts of unit inactivity make for an easier gating decision by the control logic. For instance, when the control logic of the chip is deciding whether to clock gate a branch predictor, a larger cycle-distance between dynamic branch instructions would result in more effective gating performance.

2.2.2 Technology Scaling

The amount of power dissipated dynamically during circuit activity is linearly proportional to the capacitance of the given circuit. This means that the thickness of the interconnects between transistors, and the transistor sizes themselves, directly affect the power dissipated during each switching event. In order to fit more transistors on a single chip die, the size of the technology process used continually decreases. This has the benefit of decreasing dynamic power dissipation (although, as discussed earlier, it does increase leakage). Furthermore, layout designers can use thinner transistor interconnects. This can drastically reduce power dissipation, but is eventually limited by performance degradation; a thinner interconnect means a higher resistance, and thus transition latency [27].

2.2.3 Voltage Scaling

Equation 2.1 importantly stated that the core voltage of a chip has a quadratic effect on its power dissipation. Hence, it can be seen that the biggest impact of the any modification to save power is to simply reduce the core voltage. This has been the case, and in the late nineteen-ninetees and early twenty-first century the core voltage of CPUs dropped dramatically. Unfortunately however, reducing the core voltage past 1.5 Volts causes severe problems. First of all, as discussed earlier in this section, the transistors must operate at a lower threshold. This has the effect of drastically increasing sub-threshold leakage currents. Additionally, core voltage directly affects signal propagation time in a chip; a lower voltage applies serious performance limitations.

2.2.4 Logic Optimisation

Due to the sheer size and complexity of modern processors, they are usually designed using a Hardware Definition Language (HDL). An HDL is a high level language which enables the processor designer to concentrate on high level logic issues, and avoid circuit layout considerations. A compiler for an HDL can then convert this high level behaviour specification into an actual circuit layout. This process means that there is often a detachment of the behaviour designer and the actual layout of the chip. A great deal of research has been carried out into HDL compiler optimisation techniques, and power compiler techniques, to improve the power efficiency of the resulting circuit layout [15].

2.3 Architecture & Software Level Efficiency Techniques

The important level of energy abstraction for this dissertation is at the architecture level. Before moving onto the main architecture focus (branch prediction), this

section will cover some of the other significant areas of architecture level research; both in architecture design and compiler design.

2.3.1 Activity Factor Reduction

When designing energy efficient architectures, the central aim of the designer, or computer scientist, is to reduce the activity factor of the underlying circuit. At the architecture level of abstraction, this is achieved by reducing the number of transitions that must occur to generate a particular result. This can be a different consideration to that of high-performance design: if a modification reduces the activity factor significantly enough, it can be accompanied by a performance degradation whilst still improving energy efficiency. That is to say, it is possible for something, a program for instance, to be more energy efficient even if it does not always execute as fast as it would on a high performance architecture. The activity factor is the most important term in determining the dynamic power consumption of a processor, and dependent upon the software running on the processor [117] [87] [69].

2.3.2 Delay Reduction

Having stated that even a slower architecture can be more energy efficient, it is important to note that this is not ideal. The energy actually consumed from a power source during an operation, the execution of a program for example, is the product of the activity factor and the duration of the program. This means that, ideally, a modification made to an architecture to reduce the activity factor should be made without impacting on performance; it is very easy to reduce the average activity factor of a program by simply halving the speed at which it executes [58]. This would, however, double the duration of execution, and thus likely consume more energy. This is one of the reasons that the et^2 efficiency metric gives a heavy weighting to the duration of an operation; a considerable energy cost in a processor is the general overhead of control logic through time [67].

2.3.3 Low Power Scheduling

Many modern processors have a large instruction set that contains many similar instructions. These instruction can often be used by a compiler to achieve equivalent results in different combinations and frequencies. The principal of low power (static) scheduling is to use a simple energy model of the instruction set to make decisions about how to translate high level computer programs into assembly code. For instance, in some cases it may be more energy efficient to use a series of add instructions instead of a multiply instruction, or use predicated instruction execution instead a branch. The decisions require an intelligent compiler, capable of weighing individual instruction cost against performace delay [80] [104] [79].

2.3.4 Frequency Scaling

Frequency scaling is not really an architecture or compiler design technique; it is a technique used by an architecture aware Operating System Kernel. The Kernel of an Operating System can measure the instruction throughput of a processor over time. By comparing this throughput to a known peak performance throughput, the Kernel can then reduce the hardware clock frequency that the CPU is running at to a level that is suitably matched to the current utilisation. The ideal result of this is that power is saved, because the clock frequency has a linear effect on power dissipation, and there is no performance degradation as the Kernel constantly monitors and adjusts the clock frequency when required. This reduces the activity factor of the CPU and control logic, and, when performed correctly, should not increase the delay [106].

2.4 Branch Prediction

This chapter has so far examined low-level efficiency applications. This section introduces the target subject of this project. Branch prediction is a mechanism used to overcome the possible delay introduced by machine instructions that change the control flow of a program during execution. It has become an extremely important part of almost all processor architectures [40]. Although it is not the purpose of this dissertation to act as a complete introduction to branch prediction, this section discusses the principles of branch prediction, and its significance, in terms of energy/power, to this research project. A more indepth examination of dynamic prediction techniques and their inner workings can be found in the referenced literature.

2.4.1 The Branch Problem

Modern processors use a design method called pipelining. Pipelining is used to increase the parallelism of processor architectures by breaking down the entire processor into relatively independent stages. Each stage progressively 'processes' an instruction, from fetch to execute to commit, and passes its result into the next stage. This technique enables several instructions to be undergoing execution at a given time [40]. Although this causes the execution time of an individual instruction to increase, it can dramatically increase the throughput of a processor [84]. Figure 2.1 shows an example five stage pipeline.

Different architectures break the processor down into a different number of pipeline stages. Modern high performance processors often include as many as 28 pipeline stages [5]. Breaking the processor down into more pipeline stages can further increase performance, and permits high clock speeds as the clock signal does not need to propagate through the entire processor – only through each pipeline stage.

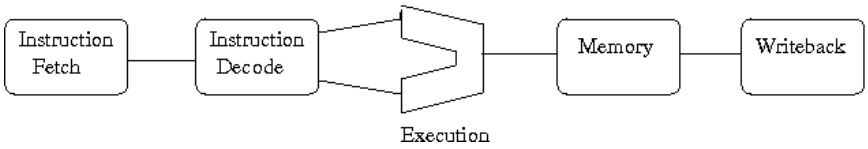

Figure 2.1: An example five stage processor pipeline

When a processor is pipelined, the problem of the branch delay region is introduced. A branch instruction can potentially change the flow of control in a program during execution based on the result of some preceding, or included, operations. In a pipelined architecture, the result of such an operation (for instance a comparison by the branch instruction) is not available until the instruction passes what is conventionally known as the execution stage. In Figure 2.1 this is shown by the third stage. The instruction fetch stage/unit must continue to fetch a stream of instructions to fill the pipeline stages behind the branch instruction (two stages in the example). These instructions could be from either path of the branch instruction – the actual direction is not known until the branch has passed the execution stage. If the processor fetches instructions from what turns out to the be the wrong path, the processor must be stalled while the incorrect instructions are flushed from the pipeline, and the correct ones are fetched to fill their places. In a high performance processor, this delay causes a significant performance penalty, particularly when it is considered that one in every six to eight instructions in the dynamic instruction stream is a branch. The branch problem is further exacerbated in MII processor (Multiple Instruction Issue – a processor with parallel pipelines), where many branch instructions may be undergoing execution at any given time [31].

2.4.2 Dynamic and Static Prediction

Several approaches have been devised to ameliorate the effects of the branch problem on performance. These approaches typically fall into the dichotomy of either a purely static or purely dynamic technique. A simple description of these two techniques is:

Dynamic Branch Prediction – Techniques which use processor hardware to dynamically (during a programs execution) decide from which branch path to fetch instructions when a branch instruction is encountered. These techniques typically base their prediction on the behaviour of previously encountered branch instructions (and their outcomes). The prediction can change each time a branch is encountered dynamically.

Static Branch Prediction – Techniques which use data available at compile time to provide a single, unchanging, prediction for each static branch instruction. The data used can either be from static code analysis, or from execution profiling data.

The best dynamic branch predictors are generally far more accurate than the best static prediction methods [83]. This is because they can better adapt to changes in datasets that affect the behaviour of branch instructions, and also the context sensitive nature of many branch instructions (the nature of the preceding basic block – there can be several).

2.4.3 Dynamic Predictors

Most modern processors, particularly the high performance variety, exclusively use dynamic branch predictors to help solve the branch problem [5]. Indeed, the focus of this project is on the existing dynamic predictor technology.

When a branch instruction is encountered dynamically it can change the execution path by changing the address of the next instruction to be fetched. Dynamic branch predictors are split into two complementary units: the first decide on whether branch will be taken or not-taken, and the second (for a taken branch) will attempt to predict the target address for the branch. If both of these predictions are correct, then the branch delay problem is solved for that encounter. Figure 2.2 shows a logical representation of how a dynamic branch predictor would look when located in the instruction fetch stage of the pipeline.

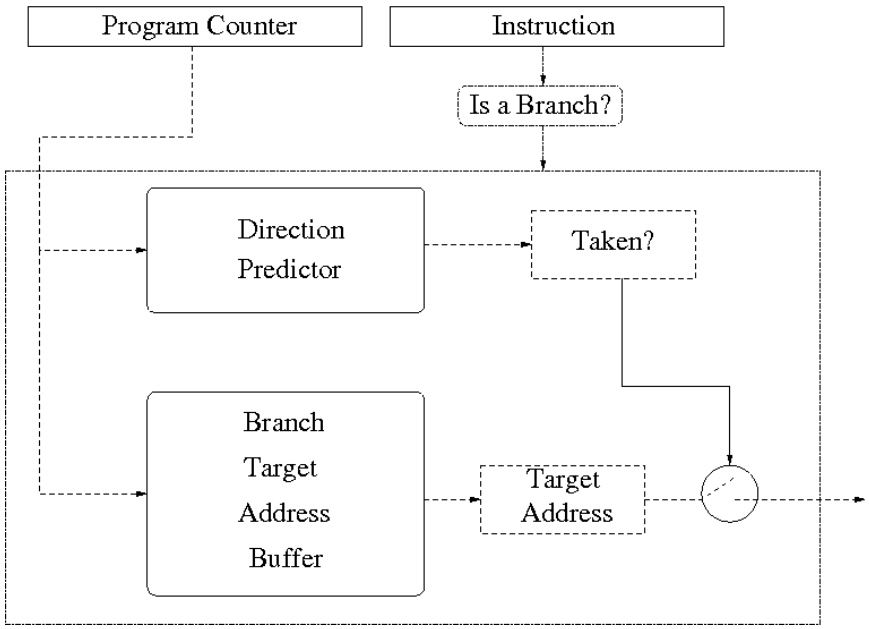

Figure 2.2: An example of a modern dynamic predictor architecture

In Figure 2.2 it can be seen how the branch predictor is segregated into a direction and target predictor, and how these units work together to produce the next fetch address for the processor.

Target Prediction

Target prediction is, conceptually, the most straight forward part of the prediction. A typical target predictor unit is a branch target buffer [84]. A branch target buffer is a small cache with a variable number of entries (typically 256-1024) which is indexed using the program counter (fetch address). The way that the program counter is used to index the cache varies depending on the implementation, but typically a hash or simple remainder-on-division operation is used. The cache can either be direct-mapped or set associative. Typically, set associativity is used since this will yield the best results when address conflicts occur.

When a branch instruction is resolved in the execution stage, its coordinating entry in the branch target address buffer is updated. Branch target address buffers tend to work very well because individual branches often have a fixed target address, or a stable target address for a relatively long period of time. When set associativity is used, the target address prediction can be very accurate when a branch instruction has been encountered once, due to the principles of temporal and spatial locality; the target of a branch tends to remain constant, at least for a relatively long period of time.

Direction Prediction

The greatest difficulty in dynamic branch prediction is predicting whether a branch instruction will cause a change of flow to its target address or not. This problem is referred to as direction prediction, and there are several hardware approaches that can be used to solve this. Many approaches were, originally, divided into two groups: local and global predictors [40]. A local predictor would use the behavioural history of a given branch to form a prediction, and a global predictor would use the behavioural history of the contextually preceding n dynamic branches when forming a prediction for the current branch. Both prediction techniques form a direction prediction in the update pipeline stage, and are updated during the execution or commit pipeline stage.

Though the focus of this project is not on the internals of a branch predictor, some of the most common dynamic branch predictors, and a brief description, are listed below:

Predict Not-Taken – This is the same as not predicting the direction/target prediction of the branch at all, and just assuming that the next instruction in the stream will be executed. This is inaccurate as most branches are taken. A similar, though more accurate, effect is achieved by predicting ‘always taken’ [40].

Predict Backward Pointing Branches as Taken – This is more accurate as many loop branches will be predicted correctly [40].

Bimodal Branch Prediction – These predictors have tables containing two bit entries. The table is indexed by some part, or all, of the program counter. Each entry holds a counter that represents a prediction of strongly taken, weak taken, weak not-taken, strongly not-taken. This entry is used to form a prediction of which branch path will be taken. During the execution stage, the relevant entry is updated by incrementing or decrementing the counter appropriately. Very large Bimodal predictors have been shown to saturate (reach their effective ceiling) at up to 93.5% correct [40].

Local Branch Prediction – Local branch predictors are implemented using two tables. The first table is indexed by some low-order bits of the program counter, and stores the taken/not-taken history of the given branch. The second table consists of a similar structure to the bimodal history table, but is indexed by the contexts of the indexed entry in the first table. This produces an appropriate direction prediction for the given branch based on its previous behaviour. Very large local branch predictors have been shown to saturate at up to 97.1% correct [40].

Global Branch Prediction – Global branch predictors use the context sensitive nature of branch instructions to predict the likely direction that will be taken on a given encounter. A basic global predictor keeps a 'shift' register, containing the history of the last n branches, which is used to index a table of bimodal counters. This simple global predictor is only slightly better than a bimodal scheme for large table sizes, and never as good as local prediction. An accurate version of a global predictor, called the *GShare* predictor, XORs the branch instruction address with the shift register, and then indexes the bimodal counter table with the result. The GShare predictor has been shown to saturate at around 96.6% correct – almost as accurate as local prediction, but can use smaller table sizes. Global prediction is easier to implement faster than local prediction as the table lookups are indepedent, but fast local implementations are possible [31].

Combined Branch Prediction – This technique uses three predictors in parallel: a bimodal predictor, gshare predictor and a bimodal-like predictor to select which of the previous two predictors to 'listen' to on a per-branch basis. Combined branch predictors are about as accurate as large local predictors [40].

Agree Prediction – A technique used in conjunction with multiple predictors to help reduce aliasing between table entries. This technique can increase accuracy, but at the cost of complexity [40].

Neural Branch Prediction – Neural branch predictors use neural network structures to form directional branch predictions. Still an emergent type of branch predictor, the main challenge is the high latency of prediction formation, but

neural predictors can become accurate more quickly during a programs execution than other prediction techniques [109].

The internals of each of these predictors are not particularly examined in this dissertation, but there are important references made to the choice of branch predictor used in the experimental architectures.

2.4.4 Power Consumption

The relative power consumption of a dynamic branch predictor varies highly depending on the size of the rest of the processor. However, in high performance architectures, the branch predictor unit will typically account for over 10% of a processors global power consumption [82]. This significant consumption figure is accounted for by the regularity of accesses and the complexity of the dynamic predictor when accessed.

A dynamic branch predictor is accessed in two situations: when furnishing a prediction in the instruction fetch pipeline stage, and when being updated with the branches behaviour in the execution/commit stage. When it is considered that around one in every eight dynamic instructions is a branch, it can be seen by branch predictors are a significant drain. Both the direction predictor and the target address predictor account for this power consumption. The ratio of power consumption between the direction and target address predictor is variable, and depends upon the configuration of both predictors. This is because a structure such as a BTB has not only a large number of entries, but also a large entry width and associativity.

The internals of dynamic branch predictors are highly optimised and have been subjected to many years of scholarly research [81]. Optimising their existing accurate and well-studied behaviour is unlikely to achieve big gains. An interesting remaining question is to what extent can the activity factor of a dynamic branch predictor be reduced without affecting its accuracy?

2.5 Summary

This chapter has shown the various mechanisms, at various levels of abstraction, that can be used to save power/energy in modern transistor based digital circuits. At the lowest level, fabrication based improvements have allowed for more energy efficient (and smaller) transistors that can be used as part of any processor design. There are also various techniques that can be used at the logic level in order to improve the energy efficiency of a designed circuit. The most relevant level for this project is the highest level of abstraction, the algorithm level, where careful design consideration can reduce the activity and time factors of a a given unit/operation.

Dynamic branch predictors consume a significant amount of global processor power [81]. The amount of power consumed is proportional to the number of accesses made to the control logic (the activity factor of the predictor). Although

this varies from processor to processor, the amount of global power consumed is always non-trivial. Dynamic branch prediction hardware has been subject to considerable research in recent decades, but this has consistently focused on improving accuracy. While accuracy (one means of improving performance) is important to energy efficiency, it is not the only variable that needs to be considered. The next chapter introduces, and discusses, the contemporary approaches proposed to increase the energy efficiency of dynamic branch predictors.

Chapter 3

Related Techniques

There are many techniques that can be used to reduce the power consumption of a dynamic branch predictor. The previous chapter discussed the broad background of power consumption and the external details of branch prediction. This chapter introduces only the most relevant contemporary hardware and software techniques for energy efficient branch prediction. The most closely related technique is the Prediction Probe Detector, and this is discussed in the most detail. The software techniques are discussed more briefly, but are expanded on in the continuation of related work in the next chapter.

3.1 The Prediction Probe Detector (Hardware)

The Prediction Probe Detector (PPD) is a cache-like hardware unit introduced into a processor to reduce the number of lookups made to a dynamic branch predictor unit. It was developed and published in 2004 by, principally, Dharmesh Parikh and Kevin Skadron at the University of Virginia [82] [81].

3.1.1 Implementation

Figure 3.1 shows the logic form of the PPD in relation to the direction and target predictors for branch instructions in the instruction fetch pipeline stage.

The prediction probe detector is a small cache consisting of the same number of lines/entries as the i-cache. It is indexed in the same manner as the i-cache, using the program counter, and produces a corresponding entry for each instruction in the i-cache. Each entry in the prediction probe detector consists of two bits; one bit controls accesses to the direction predictor and the other bit controls accesses to the branch target buffer. The two bits are used to avoid accessing the dynamic branch predictor logic for non-branch instructions, and the direction predictor logic

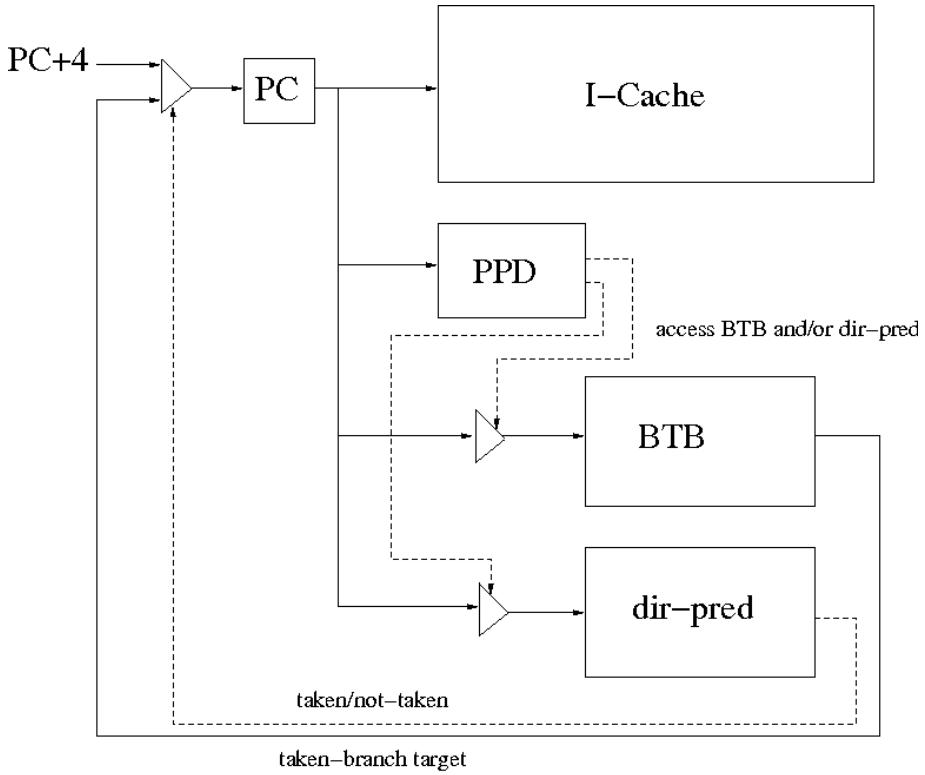

Figure 3.1: The Prediction Probe Detector

for branches that are unconditional. The entries in the PPD are updated with new predecode bits during an i-cache miss (when a new entry is fetched), and reflect the required branch predictor access levels.

The PPD avoids having to access the dynamic branch predictor logic for non-branch instructions in a high performance environment by using the properties of a small cache to permit a parallel access of the branch predictor with the i-cache; in highly clocked processors, it is not always possible to wait for the i-cache fetch to complete before accessing the branch predictor. Instead of having to access the entire branch prediction logic in every cycle, the processor now needs only to consult the relatively small PPD unit, which will then dictate if a branch predictor access is necessary. In processors with a slower cycle time (lower clock rate), a parallel branch predictor access is not always required, and hence a proportion of this effect can be achieved with simple pre-decode logic.

The PPD was shown to conserve, on average, 3.1% [82] of global processor power. This result is significant, but the introduction of additional processor logic clearly introduces additional power dissipation into the processor (although this was accounted for in the study).

An extension to the PPD was then implemented that allowed the PPD to recognise highly biased, or 'unchanging', branches. Never taken branches were represented using the existing two bits in the PPD entry (by disabling access to both the direction and target predictors). Highly-biased taken branches require an additional bit in the PPD that is used to assume a taken direction prediction. The implementation of this was not made completely clear in publication, however, it seems that an experimental implementation involved using profiling data to mark commonly taken/not-taken branches in a compacted trace. This trace was then read by the processor simulator at execution to set the additional bit(s) in the PPD. This is not a plausible implementation, but did show encouraging results: an additional global processor power saving of 2%. It is unclear how the decision was made as to whether a branch was heavily biased or not, and how many branches were marked in this way. Any errors incurred by the assumed branch behaviours are simply a new source of branch misprediction.

3.1.2 Pipeline Gating

Pipeline gating is a technique implemented alongside the PPD to further save power consumed executing misspeculated branch paths [82]. Pipeline gating works only in conjunction with certain branch predictors, and assigns each branch either a high or low confidence estimation based on the coherence of the two internal predictions of the hybrid predictor. Assigning a confidence to a branch prediction is difficult [39] [56]. When the number of sequentially fetched low-confidence branch instructions exceeds a specified threshold, the pipeline is stalled as it is considered likely to be executing on a mispeculated path. Each time a low-confidence branch is resolved, the counted number of low-confidence branches is decremented.

Pipeline gating is not discussed in great detail here, since it was found to have very poor power conservation potential (much less than was suggested by previous studies). Pipeline gating introduces significant logic into the processor itself, and erroneous pipeline gating behaviour can potentially impact heavily on performance and power consumption. Additionally, as the pipeline depth increases, pipeline gating becomes less effective. Finally, pipeline gating can be implemented irrespective of other branch prediction techniques being used, and so isn't on the critical path of this investigation.

3.2 Software Based Approaches

There are many software based approaches that have been used to increase the energy efficiency of microprocessors. The general concepts of these methods were discussed in the previous chapter. However, none of these methods are explicitly designed to exploit the potential energy savings available in the dynamic branch predictor logic of a processor. The next chapter investigates, and discusses, some older applicable software methods, but the next subsection introduces the most prominent and investigated technique in use.

3.2.1 Hinting and Hint Instructions

Branch prediction, as discussed in the previous chapter, can generally be achieved using two methods: static prediction or dynamic prediction. Static prediction assigns a prediction for a branch at compile time, which is interpreted by the hardware, and dynamic prediction introduces hardware logic that will produce a prediction based on the dynamic behaviour of the branch(es). When considered separately, dynamic branch prediction is found to have significantly higher accuracy. However, this is not the case for all branch instructions, but rather the program as a whole.

The principle of hinting, to conserve energy, is to assign a static prediction to only certain branches, with the aim of reducing the number of accesses made to a dynamic branch predictor. The hint is some way of communicating the likely direction of the branch to the dynamic hardware. This should, ideally, have the benefits of dynamic accuracy, but without all of the costs in terms of energy.

Assigning such static predictions, with the aim of conserving energy, has been experimented with in a variety of ways. Some studies, implemented in VLIW processors have introduced a new instruction, called a hint instruction, which is inserted with the instruction word in front of a branch to inform the hardware of an upcoming branch instruction [76], and that a prediction must be furnished. This avoids the need to predict for non-branch instructions. Static predictions can be assigned using either compiler heuristics or profiling. Profiling is the most accurate, however it is limited in previous studies [70] by the method used to decide whether

a branch should be marked as biased or not; generally a fixed bias threshold is used, and if this is exceeded, then the branch is assigned a hint [35]. This does not work well with dynamic predictors as it is possible a biased branch could still be predicted better by the dynamic predictor (some branch patterns can be predicted with almost total accuracy).

An implementation of compiler based VLIW hint instructions, to avoid direction prediction for some biased branches, was demonstrated by a research group at Politecnico di Milano to reduce the number of branch predictor accesses by up to 86% [70]. However, no reliable indication was given of the possible power savings, and these results were for selected benchmarks only (not a full suite). It is possible that the results published were not fully representative of a cross section of tasks.

The principle of the hint instructions has also been further extended to provide a simple direction prediction for heavily biased branches, or branches which will never been taken. These static direction predictions are relatively crude and were assigned based on compiler heuristics that decided on how likely biased a branch is. Again, these static predictions only removed the burden of direction prediction, not target prediction.

3.3 Analysis and Summary

The most closely related and significant work in the field of energy-efficient branch prediction is the Prediction Probe Detector. This is a hardware unit that is used to store information about which cache lines contain branch instructions and, for certain branches, whether they should be assumed taken or not taken. Previously investigated software methods of energy-efficient branch prediction take a similar approach, but use hint instructions in VLIW instruction bundles to communicate the same information.

There are several key limitations with both approaches:

1. The focus of energy saving is on the lookup of a branch predictor in the instruction fetch stage, when furnishing a prediction. This is important, but all committed branch instructions will also update the branch predictor, which is equally, if not more, costly than the lookup phase, in terms of energy. The update phase has possibly been ignored [82] due to the lack of reliable static prediction assignments for many branches (the focus was on not predicting for non-branches, in highly clocked processors).

2. The assignment of any static prediction to a branch was achieved using rudimentary methods. Branches were given a statically predicted direction based on whether they appeared ‘likely’ to be biased.

3. Neither technique makes use of any of the existing methods of branch removal/avoidance and scheduling techniques. These are discussed in further detail in the next chapter.

In additional to the problems stated above, the PPD also introduces significant hardware. While this hardware was accounted for, and did save global processor energy, the actual cost in power was not specified. The PPD is the size of a small cache and would hence have significant power and timing implications. The hinting methods used in previous software efficiency studies have principally focused on VLIW processors (due to the availability of instruction slots in instruction bundles).

The area of energy-efficient branch prediction is not well investigated at the level of the hardware software interface. It is likely that there is room for improvement if hardware and software techniques can be used in cooperation to achieve a more successful result. This has been considered in the performance arena [83]. The next chapter progresses to discuss some early stage investigations into branch prediction and energy, and also discusses more of the existing software branch avoidance techniques.

Chapter 4

Initial Investigation and Preliminary Research

The previous chapter has introduced the key related work on energy-efficient branch prediction. Previous work [82] has shown that, in order to save power during a programs execution, the key variable to be optimised is the number of accesses made to the dynamic branch predictor. This chapter investigates several other existing methods that could possibly be used to remove the number of accesses to a dynamic branch predictor. Their initial use was not for energy efficiency but, as discussed in this chapter, it is possible to make use of them to reduce the load on a dynamic branch predictor.

4.1 Research Question Focus

The existing work in the area exhibits the typical dichotomy of static versus dynamic methods with little handshaking between the two. The main research question being addressed in this project is whether it is possible to use a combination of dynamic branch prediction with static methods to achieve a more beneficial result, in terms of power, than using the two paradigms in isolation. To properly achieve this goal, it is necessary to conduct a review and critique of existing static methods of branch removal and prediction, and examine how these could potentially be used.

Dynamic predictors have been extensively researched and can achieve a very high degree of accuracy. It is unlikely that modifying their highly optimised composition would achieve any great results in terms of power. The question, and goal, of this project is to evaluate a possible method whereby the activity factor of a dynamic branch predictor can be reduced, and thus conserve power/energy.

4.2 Static Methods to Avoid Dynamic Branch Prediction

Dynamic branch predictors are not always a popular method of handling the branch delay problem in pipelined processors. Indeed, when silicon space on in a processor is at absolute premium, and the fabrication process does not permit any additional complex logic, several different methods for statically removing/predicting the behavioural outcome of a branch can be used. Although fabrication technologies and feature sizes now readily permit units like a dynamic branch predictor, it is useful to review the existing static methods and evaluate how useful these methods may be when applied to reducing the power consumption of a dynamic branch predictor.

4.2.1 Delay Region Scheduling

The branch delay region, as discussed previously, is a number of cycles during which the outcome (direction and target) of a branch instruction has not been calculated. The number of 'slots' for instructions in this region typically equates to the number of pipeline stages between the instruction fetch stage (IF) and execution (EXE) stage, inclusively.

Delay region scheduling is a static process whereby the compiler tries to move an appropriate number of instructions into the delay slots following a branch instruction in the static code [84] [40]. Rather than the processor dynamically taking any action upon encountering a branch in the dynamic stream, the processor simply ignores the fact that a branch instruction may affect the control flow of the program, until the execution stage. The instructions immediately following the branch are sequentially fetched as normal. In the execution stage, when the target of the branch instruction is known, the program counter is updated to reflect the appropriate target of the branch.

The method used by delay region scheduling implicitly requires that all instructions moved into the delay region are independent of the outcome of the branch instruction dynamically. There are two approaches to this, which are discussed here. Both approaches are similar, but move candidate instructions into the delay region from different types of locations. They are generally used as separate exclusive methods and rarely in conjunction with one another.

Local

Scheduling a branch delay region 'locally' refers to moving branch independent instructions into the delay region from the *same* basic block as the branch. This means the instructions precede the branch in the static code.

Instructions can be moved into the delay region from the same basic block when they do not modify registers that either directly, or indirectly, affect the direction or target of a branch instruction. The compiler steps through each instruction

in a static basic block in reverse, starting from the branch, and checks for direct or indirect branch dependencies. If there are no dependencies then an instruction can be scheduled (moved) into the delay region. This process is discussed in more detail in the next chapter, and then in practical detail in the simulation chapter.

Local delay region scheduling has the advantage that, where the delay region can be filled, it is always a 'win'. It is relatively simple to implement and requires little in the way of hardware logic. However, if a delay region cannot be filled for a given branch, the (remaining) delay slops must be filled with NOP instructions which essentially waste processor cycles. Processors with shallow pipelines, and thus small delay regions, are ideal candidates for the use of the local delay region as it is easy to find candidate instructions to scheduling into the delay region. However, even in this case, using local delay region scheduling as an exclusive branch resolution mechanism is increasingly difficult with modern compilers that highly optimise code and tend to make branches highly dependent on closely preceding instructions.

Global

Global delay region scheduling is a process where the delay slots for a given branch are filled with instructions from *other* basic blocks. The 'other' source for candidate instructions can be chosen by two methods:

1. A statically predicted target basic block
2. Analysing the directed graph structure of basic blocks to discover independent instructions from succeeding basic blocks that are *always* executed after the current basic block

The first method requires some way to decide which basic block, of the two possible targets, would be most profitable to schedule from. The simplest way is to assume that forward pointing branches are taken as this is statically the most accurate simple heuristic. However, a more accurate way is to use profiling data to find the most commonly taken dynamic direction for the branch in question. The instructions do not need to be branch independent; simply the first N instructions are selected to fill the delay region (as this is a form of static speculation).

The second method requires very complicated static analysis by the compiler. This process can be very difficult in modern optimised code for the very same reasons that plague local delay region scheduling.

Additionally, both methods of global delay region scheduling implicitly cause code expansion. This is because a basic block can have multiple entrants. Rather than actually moving the instruction statically, the appropriate instructions are copied from the target basic block into the delay region, and a new entry point is created in the target basic block. The delay scheduled branch in question is then changed to point at the new entry point, while other entrants to the target basic

block use the original entry point (as they have not been scheduled with the same instruction). The copied instructions result in code expansion as they still exist in the target.

A simple example of this is shown below for a processor with three delay slots:

```
SCHEDULED_BRANCH_BLOCK:
INS1
INS2
INS3
INS4
BRANCH: TARGET_BLOCK'
T_INS1
T_INS2
T_INS3


TARGET_BLOCK:
T_INS1
T_INS2
T_INS3
TARGET_BLOCK':
T_INS4
T_INS5
T_INS6
```

The globally scheduled delay region can be used for all branches, but has the problem that it requires hardware support when speculatively scheduling from a target basic block. The delay region, in case number two, is scheduled speculatively and thus requires some way for the hardware to throw away falsely executed instruction, and then recover. This is usually accomplished using a hardware unit such as a reorder buffer.

4.2.2 Static Prediction and Instruction Hints

Static branch prediction, using instruction hints, is a method of assigning an unchanging prediction to a given branch, and this prediction is then used dynamically

by the processor to decide which direction to assume for the branch. Static branch prediction has been used either as a stand-alone method of branch prediction, or to override the dynamic predictor [51] [47]. The processor, during the instruction fetch stage, will examine the hint-bits in a branch instruction (usually in a fixed location for logical simplicity) and determine which direction to fetch instructions from.

The key difficulty in static branch prediction is deciding which direction to assign as a static prediction. There are two main approaches:

1. Use compiler heuristics and analysis to determine, from the context of a branch instruction, whether it is likely for part of a loop construct and thus be iterated many times and be strongly taken

2. Use dynamic profiling data too determine which branches are strongly taken and which branches are strongly not-taken

The most accurate method tends to be profiling as it takes into account the dynamic behaviour with a target dataset [42].

Static prediction with instruction hints however has one key difficulty: while it provides the likely branch direction, it does not provide the target. This is resolved in different ways depending on the architecture [51] [5]. The simplest solution is to introduce a hardware unit such as a branch target buffer which only provides the target address of the branch on a hinted-taken branch. This usually works well as the branch targets are commonly unchanging. However, it introduces significant hardware. Another option, such as that used by the PowerPC architecture, is to design the instruction set orthogonally, such that it is very simple to calculate a target address in the Instruction Fetch stage using very simple decoding.

Static branch prediction fares poorly as a complete prediction mechanism in modern processors. This is because a dynamic branch predictor is almost always more accurate as most branches are not completely biased to one direction. Using static predictions to override a dynamic prediction is usually fruitless when considering this accuracy, and only has benefit when passing a hint which advises the processor of a very difficult-to-predict branch.

4.2.3 Guarded Execution

Guarded Execution is another method that can be used to overcome the branch delay problem, and is often also referred to as conditional or predicated execution [98] [53] [97]. Guarded instructions are special instructions in the instruction set that mimic a small subset of existing committal instructions, such as move, add, load and store, but provide a built in condition detection. This condition will take the same form as that evaluated by a branch instruction. Instructions from a target path can be scheduled into the delay region and assigned to guarded instructions.

If the conditional guard on the instruction is true, the processor commits the instruction's result. If the conditional guard is false, then the instruction is treated as a NOP instruction, and no value is committed.

Successful utilisation of guarded execution scheduling will remove the need for dynamic prediction for branch instructions, but the benefit, in terms of both performance and power, are contingent on the compiler successfully deciding which branch path to schedule from; this decision can be made in a similar way to global delay region scheduling. In some situations, such as small loops and simple control constructs, guarded execution can completely remove a branch instruction (up to 33% of branch instructions can be removed [97]).

The disadvantages to guarded execution are that it puts pressure on the register file (in complex control structures) and is limited in accuracy in the same way as global delay region scheduling: although it provides a method to avoid dynamic prediction, it is likely to cause a performance penalty when instructions behave as NOPs after poor compiler scheduling.

4.3 Hardware Multithreading

Hardware multithreading refers to an architecture design that has support for following multiple threads of execution in parallel. It has been argued that this architecture design approach could be used to resolve the branch delay problem by following both possible paths of a branch instructions. However, this approach will, by definition, give the equivalent of only a 50% prediction accuracy. In terms of both performance and power, this is useful only for branches that are particularly difficult to predict. Furthermore, the parallel execution units can almost always be better utilised when left to follow multiple threads, as opposed to speculatively executing both paths of a branch instruction.

There are projects that use a large number of parallel units to achieve large scale parallelism and, within this scheme, schedule different threads onto processor cells while a thread is stalling in wait of branch resolution. One project that takes this approach is called microThreading [57]. The problem with this approach lies in its radical nature; it is more desirable, for this project, to have a smaller architecture impact in any approach used to reduce branch predictor power consumption in embedded processors.

4.4 Initial Experiments

4.4.1 Removing Dynamic Branch Predictors

The first, and most obvious, suggestion that is often made when considering the power consumed by a dynamic branch prediction unit is to remove the dynamic

branch predictor completely. Indeed, a dynamic branch predictor was originally conceived as a device to increase the performance of high speed processors [31] [84] [40]. It it is also true to say that in the embedded, power sensitive field the performance, or execution time in seconds/cycles is not necessarily the metric that is being optimised. Hence, a logical consequence of this often seems to be the outright removal of a dynamic branch predictor: if we don't care so much about performance, and a branch predictor uses significant power, then perhaps it should just be removed?

The dynamic power cost of a branch predictor, often over 10% of the processor's global power consumption [81] [82], is, in fact, offset by its power savings in the number of cycles it reduces execution time by. This is because, when optimising for energy efficiency, a key part of the metric is the amount of time a given operation, or operations, take to complete. While a dynamic branch predictor may use 10% of global power on a per-cycle basis, this does not take into account the global power change. The total power used by a processor with a dynamic branch predictor unit is, for example, equal to global power multiplied by the number of cycles that the given program executes for. Simply removing the branch predictor fails to take into account its important affect on the number of cycles that a program executes for. The branch predictor will invariably reduce the total execution time of a program.

Processor cycles are very important in energy efficiency. In each cycle, a vast amount of logic is 'clocked' (a clock signal is propagated around the processor logic) and various units within the processor will change state. Clock signals, and state changes in general, are very expensive in terms of power. If these state changes are occurring on redundant instructions, such as misspeculation or pipeline 'bubbles', then energy is wasted on instructions that will never be committed. In this sense, it could well be the case that a branch predictor is actually a power *saving* device.

A small experiment was carried out using the EEMBC benchmarks and and the HWattch processor power simulator. Both of these are discussed in the simulation chapter of this report, but are referenced here for completeness. The experiment used the following general baseline:

- Scalar Processor
- Seven stage pipeline
- Separate instruction/data cache
- Full execution of the entire EEMBC benchmark suite
- With Branch Prediction: a large local predictor (1024 entries)
- Without BP: assume not-take (fetch the next instruction as normal – PC+4)

A more detailed specification of this baseline can be seen in the baseline section of the experimentation chapter later in this report (7.2); however, for the basis of this small experiment, the above information is sufficient to demonstrate the result.

The outcome of executing the entire EEMBC suite on the baseline shown was that, using a weighted average, global power consumption increased by approximately 10% when the branch predictor was *not* present. This result confirms similar experiment conducted elsewhere [82].

The importance of this result is two-fold: firstly, it shows that a dynamic branch predictor is actually a power-saving device and cannot simply be removed, and secondly, it demonstrates that any attempt to decrease the power consumption of the dynamic branch predictor logic must not come at the cost of a significant change in accuracy.

The experiment shows that reducing the accuracy of any existing form of branch prediction will, at some stage, have an extremely negative observable effect in terms of power.

4.4.2 Instruction Stream Research (HTracer)

In the early stages of research, a dynamic instruction tracer tool was developed called HTracer (Hertfordshire Tracer) [41] [45]. HTracer is capable of producing selective traces of the dynamic instruction stream on any platform for which the Linux operating system is available. This is achieved by close coupling the Linux kernel calls, and with a flexible system of specifying which kinds of instructions to trace. A more detailed description of the capabilities of HTracer can be seen in the eponymous publication contained in Appendix A.

Extensive branch traces were produced for the execution of various isolated benchmarks from both the EEMBC and the SPEC 2000 benchmark suite on the PowerPC architecture. These were analysed to discover the extent to which branches are biased to a particular direction, and which kinds of branches tend to account for a large proportion of the dynamic instruction stream.

The results of the trace analysis showed that a large proportion (around 70%) of the dynamic instruction stream was made up of a single branch format/type: the offset type branch. This is logical, when considered, because offset branches are used by relatively small, highly iterative loop constructs. These loops typically execute a large number of times and would therefore likely account for a large proportion of the branches in a given instruction stream. Nevertheless, this result is useful as it verifies the behaviour of a 'real' instruction set.

The results gathered from the instruction traces also showed that a high propertion of branches are very biased to one direction:*in many programs, up to 70% of all dynamic branches encountered are more than 85% biased to one target direction* [43]. Branches that are so greatly biased are very easy to assign a static prediction to, and it is possible that this static prediction could be as accurate as a dynamic prediction for these branches.

4.4.3 I-Cache Experimentation

Additional experimentation was carried out with respect to the I-Cache and the dynamic branch predictor [32]. The Wattch power analysis simulator was modified to include two special additional bits in each line of the instruction cache. These bits represented saturation information for each branch in the I-cache. The bits were set based on the branch history stored in the directional predictor history tables, and were used to bypass a dynamic prediction for very biased branches. When a branch exceeded a certain level of bias to one direction, the bits would be set to avoid accessing the dynamic branch predictor the next time(s) the branch is encountered.

Although this could only be used to avoid accessing the direction part of the dynamic branch predictor, and not the branch target predictor, it did show that it is possible to save some power without significantly affecting branch accuracy. Introducing the additional bits saved around 1% of global processor power. Another clear downside of this particular approach is that it increases the size and complexity of the cache, and this is already a sizeable and complex processor unit. More details of this implementation can be found in the publications appendix.

4.5 Summary

This chapter has introduced and discussed various methods that have been previously used to avoid the need for dynamic prediction. The first experiment described in this section revealed that, in fact, a dynamic branch predictor is actually a power saving device in most contexts. This means that a dynamic branch predictor saves global processor power by reducing both the number of misspeculated instructions executed and also by the avoidance of redundant cycles following a misprediction. This is significant as it shows that any power saving modifications to a branch prediction paradigm must not come at the cost of branch prediction accuracy; misprediction is expensive in terms of both performance *and* power.

Delay region scheduling is an interesting concept as it, in theory, completely removes the need of a dynamic predictor. However, the reality is that it is very difficult to schedule into the delay region effectively. Local delay region scheduling is the most beneficial, but extremely difficult to find candidate instructions for, particularly in contemporary optimised assembly code. Global delay region scheduling has the advantage of making it easier to find candidate instructions, but introduces static speculation and code expansion.

Guarded instruction execution is an interesting concept, but suffers from similar scheduling problems to the global delay region, and also applies pressure on the register file for the various guards required in complex control sequences. Hardware multithreading is an inappropriate method of resolving the branch problem for embedded processors as it is both energy-inefficient and inaccurate.

Preliminary research has shown that an ideal solution might be to use a com-

bination of the static methods discussed in this chapter in conjunction with an existing dynamic prediction paradigm. The static methods should be used on highly biased branches, or those which can be scheduled with no speculative cost. Such an algorithm could complement existing dynamic predictors by reducing the number of dynamic accesses, and thus reducing global power consumption over a program's execution. A formulation of this ideal was conceived which combined local delay region scheduling and profiling data (in the form of hint-bits). This 'combined algorithm' is discussed in great detail in the next chapter, and leads into an in-depth investigation.

Chapter 5

The Combined Approach

The combined approach makes use of two traditional processes, with some novel modifications, and uses them in conjunction with the architecture's existing dynamic branch predictor. The impetus behind this approach is to represent runtime information statically in a branch instruction. This can then be used by simple hardware to bypass the need to access the dynamic branch predictor for as many branches as possible. The most important goal of this approach is to have as little impact on the overall prediction accuracy of a program as possible. Hence, the algorithm should not increase the execution time of a program, or cost additional power through mispredictions.

5.1 Local Delay Region Scheduling

In contrast to scheduling into the delay region from a target/fallthrough basic block, a locally scheduled delay region takes branch independent instructions from the same basic block that precede the branch in the static code.

A branch independent instruction is any instruction whose result is not directly or indirectly depended upon by the branch to calculate its own behaviour. Moving a non branch independent instruction into the delay region would affect program semantics [80] [40].

Deciding which instructions can be moved into the delay region locally is straightforward. Starting with the first instruction from the bottom of the given basic block in the static stream, above the branch (the branch instruction's predecessor), examine the target register operand. If this target register is NOT used as an operand in the computation of the branch instruction then it can be safely moved into the delay region. This process continues with the next instruction up from the branch in the static stream, with the difference that this time the scheduler must decide whether the target of the instruction is read by any of the other instructions

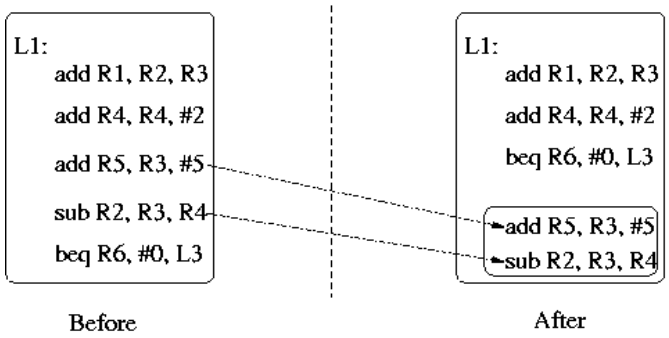

Figure 5.1: An example of local delayed branch scheduling

below it (which are in turn used to compute the branch). This means the operands and target must be checked for data hazards [33]. Figure 5.1 shows how the two instructions from basic block 'L1' have been moved into the delay region of the branch and will always be executed (and usefully committed) in the stages of the pipeline behind the branch. The algorithm used is explained in more detail in the next chapter, which discusses its implementation. In fact, the way in which the algorithm is used in the implementation greatly simplifies the dependency checking, so this is deliberately given a curt discussion here.

Local Delay Region Scheduling is an excellent method of utilising the delay region where possible; it is always a win, with no penalty and completely avoids the use of a branch predictor for the given branch. This is true because we can be certain that the instructions moved into the delay region will always need to be executed; by the time the processor needs to fetch instructions from a target path, the target of the branch has already been calculated. Scheduling in this way avoids any speculation at all for the given branch and requires no control logic to recover from instructions that were scheduled from the wrong path of a branch.

The clear disadvantage with local delay region scheduling is that it cannot always be used. There are two causes that result in this:

1. In well optimised code, it is difficult to find branch independent instructions in the same basic block that can be moved into the delay region.

2. In deeply pipelined processors, the delay region can be vast (e.g. 8 clock cycles before branch resolution. This means finding 8 instructions within the same basic block to the fill the delay region, which is extremely hard). For a branch where it has been chosen to use the local delay region, any slots which cannot be filled with useful instructions must be padded with NOPs. This is wasteful.

However, the locally delayed branch is still very useful for unconditional absolute branch instructions, particularly in embedded processors. This is because an unconditional absolute branch instruction is not dependent on any preceding

instructions when calculating either its target or direction. This means that the number of delay slots that can be scheduled is limited only by the size of the basic block from which instructions are being scheduled. Furthermore, in an embedded processor, the number of branch delay slots is much lower than desktop machines; typically is it between 2 and 5 instructions (depending on the architecture).

Although the local delay region is no longer a useful method on its own, it will prove profitable when used in combination with the methods proposed in the following sections [46].

The Delay Region in SuperScalar Processors

The delay region in superscalar processors, which use dynamic scheduling algorithms, can be of variable size. This means that it is possible there is no fixed number of delay region slots to schedule into. However, this can be overcome by using profiling data. Each branch instruction exists within a static context; during execution there is a fixed number of entrants to a given basic block. The profiling data can be used to determine the minimum number of delay slots available for a given branch, and also the maximum, over an adequate dataset.

The delay region is scheduled with instructions to fill the minimum available delay slots and then 'NOP' instructions are inserted up to the maximum number of slots. This ensures program semantics are always maintained and allows the use of the local delay region in a superscalar environment. However, this process does rely on a dataset 'touching' each area of a program sufficiently. Typically, particularly in shallow pipelines, the minimum and maximum numbers of delay slots available for a given branch are extremely close (if not the same). In scalar processors, which currently constitute the majority of embedded processors, there exists no such problem.

5.2 Profiling

The central aim of the combined algorithm is to associate an accurate static prediction with as many branches as possible, so as to reduce accesses to the dynamic branch predictor (in order to save energy). This can also be achieved through static analysis of the assembly code of a program; it is often clear that branches in loops will commonly be taken and internal break points not-taken. However, such a method does not take into account either the behaviour or accuracy of a coexisting dynamic predictor.

A more reliable method is to observe the behaviour of a given program, at the assembly/machine level while it is undergoing execution with a sample dataset or datasets [41]. This means that the behaviour of each branch instruction can be recorded in the form of a specially adapted program trace which logs a detailed history of selected instructions while undergoing execution. Any relevant infor-

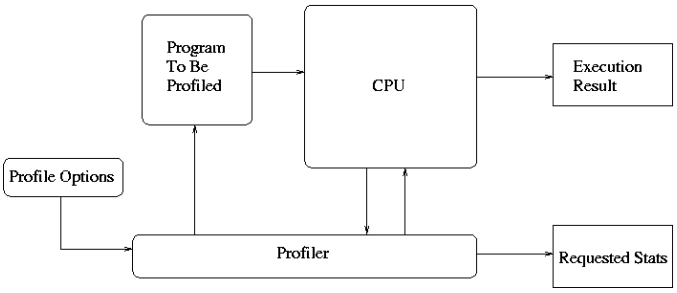

Figure 5.2: The basic structure of profiling

mation about a branch can be extracted and used to form static predictions where possible. This is the essence of profiling in the context of static branch prediction. The number of datasets that any given program is profiled with will affect the accuracy of the profiling results. The more diverse datasets used, the more widely applicable the results will be, but care must be taken in selecting only datasets that represent the target execution domain.

Profiling derives its key abilities from the fact that a program will behave similarly across datasets in the same domain. This means that, for instance, profiling a JPEG compression program with one image dataset will yield results that can predict the behaviour of the program with another image dataset. This is particularly useful for branch instructions as they commonly only behave differently across datasets at a few key moments during the programs execution.

5.2.1 Assigning a Static Branch Behaviour

The key limitation of the traditional use of profiling to form static branch predictions is that it can only be used to show the bias of any given branch instruction. While this is useful for assigning a static branch prediction in a processor without a dynamic predictor, it has a degrading effect on overall prediction accuracy when used in conjunction with a dynamic predictor, and in turn actually increases power consumption [82].

This limitation arises when one considers how profiled data about the directional history of a branch instruction undergoing execution could be used to assign a static prediction. In a static-only prediction model, a branch instruction is assigned a prediction to instruct the processor to assume the path of profiled directional bias; it does not matter if the bias is only 1% (51%) to a specific direction as this will still be the best prediction to make.

However, consider the situation where there exists a dynamic branch predictor in the target processor, and we are trying to reduce accesses to this without reducing overall branch prediction accuracy. It needs to be decided at what bias level a branch is assigned a static prediction, but this is difficult since it is unknown whether the dynamic predictor could achieve a higher accuracy than a static as-

signed prediction for a given branch. Given the general accuracy of dynamic predictors, it is clear that a fixed static prediction bias assignment level is likely to be either:

1. Set so high that no branches are removed from dynamic prediction or
2. Set low enough to decrease dynamic prediction accesses, but also then often removes branches that are better left for dynamic prediction, thus increasing power consumption by the increased delay of mispredictions

5.2.2 Adaptive Branch Bias Measurement (ABBM)

The solution to this problem requires additional information to be recorded during profiling execution. Instead of just storing the directional history of a branch instruction, the profiler should also log the associated predicted direction of the dynamic predictor for the given branch. Using the full directional and dynamic prediction trace of a program, a static prediction assigner can build up an individual profile for each branch which shows the dynamic prediction accuracy for a given branch verses its bias. This information can then be used to assign a static branch prediction to only those branches whose dynamic prediction accuracy would not be impaired by a static prediction – when a given branch's bias is greater than or equal to its dynamic prediction accuracy. It is corollary to say from this, that the more accurate the dynamic predictor in use is, the fewer branches will be removable with a static prediction.

5.3 The Combined Algorithm

The combined approach is a method designed to make use of the strengths of both the local delay region and the novel Adaptive Branch Bias Measurement (ABBM) through profiling. When these two methods are combined, it is expected that the number of dynamic branch predictor accesses can be radically reduced due to the principles of execution locality.

The information that needs to be conveyed in a given static branch instruction, in order to avoid accessing a dynamic predictor, is as follows:

1. Statically predict taken. Do not access, or update, the dynamic predictor for this branch.
2. Statically predict not-taken. Do not access, or update, the dynamic predictor for this branch.
3. Use the locally scheduled delay region. Do not access, or update, the dynamic predictor for this branch.
4. Use the dynamic predictor (and update it).

It is important to remember that two accesses need to removed for a given branch: the access to furnish a prediction and the access to update the branch predictor. The representation of this information requires two bits in a branch instruction (these bits are now referred to as hint-bits) and is discussed in further detail in the next subsection.

Algorithm 1 shows a more precise definition of how the combination of the two techniques should work. The algorithm relies on only setting the profiled hint-bits if the branch is as, or more, biased than its prediction accuracy. This method is novel because with an adequate data set(s) it should have minimal impact on the prediction accuracy for the given branch instruction, and thus not increase the execution time of the program as a whole (something which would actually likely result in an increase in power consumption).

```
Input: All Assembly Files of Programs
Output: Appropriately Hinted Assembly Files
foreach Program do
    foreach Assembly File do
        foreach Branch Instruction do
            Initially, set hint-bits to "Use the dynamic predictor for this
            branch"
            if Branch == Unconditional Branch then
                Set hint-bits to use local delay region and move two
                instructions preceding branch into delay region (if possible)
            else
                if Branch's Profiled Bias ≥ Dynamic Branch Predictor's
                Accuracy for this Branch then
                    Set Hint-Bits to Predict Profiled Bias
                end
            end
        end
    end
end
```

Algorithm 1: Combined Dynamic Branch Prediction Reduction Algorithm

Figure 5.3 shows how Algorithm 1 would work in the familiar setting of compiling a program by introducing additional stages into the Gnu Compiler Collection Chain.

5.4 Hardware Implementation

In order to save energy/power in a particular processor architecture, the hardware needs to include some logic to take advantage of the four hint cases described previously. Fortunately, these hardware modification are relatively simple, but there

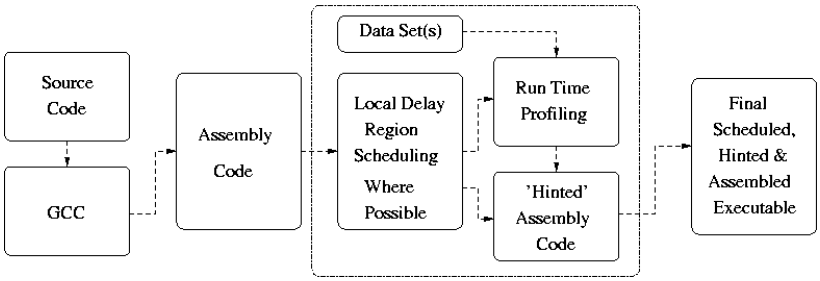

Figure 5.3: Block model of the profiling and hinting regime

are some subtle considerations that are required in order to gain the maximum benefit from the combined algorithm. The following section discusses instruction set modifications in further detail to clarify those points that have already been discussed.

5.4.1 Instruction Set Modifications

The simplest way to represent the four behaviours required by the combined algorithm is to use hint-bits. Hint-bits are additional bits contained within an instruction, or instructions, in a given instruction set. In the case of avoiding accessing a dynamic branch predictor, there is a decision to be made about whether to include the behavioural hint-bits in all instructions or just in branch instructions. Many modern processors already predecode instructions to determine whether to access the dynamic predictor unit (they only access the dynamic predictor on a branch instruction); in which case, hint information need only be included with the branch instructions themselves. Including hint information in branch instructions alone is much more acceptable since branch instructions typically have at least two redundant bits. However, it could easily be the case that a designer may wish to use the hint-bits themselves as branch predecode information, and thus implement them throughout the entire instruction set. Some modern embedded instruction sets [51] already include hint-bits in branch instructions, but they are currently only used as a fallback and no compiler/hardware makes use of them as an energy saving method. The model proposed in this study includes hint-bits only in the branch instructions as almost all energy efficient architectures already check to see whether an instruction is a branch by using predecode logic and an appropriately designed instruction format.

The most important point for consideration in the instruction set, and for compilers, occurs when the hint-bits are required to represent an ‘assume taken’ branch. As described in the Chapter 2, the energy cost of accessing a dynamic branch predictor is split between the direction predictor (pattern history tables) and the target address predictor (branch target address buffer). In order to make energy saving as effective as possible, accesses to both parts of the dynamic predictor must

be avoided whenever possible. The target address of a branch instruction is not known (at the earliest) until the instruction is decoded. This makes avoiding the use of the branch target address buffer difficult without simply replicating branch instruction decoding in the instruction fetch stage of the pipeline. In an energy sensitive environment, this is counterproductive.

Fortunately, there does exist a simpler way to work around this problem. Most instruction sets, particularly RISC variants (the most common type in an energy sensitive environment [51] [7]), overwhelmingly use a particular branch instruction format. Although a large variety of branch instructions are usually available, the majority of branch instructions actually used are offset type branches that are used to jump a short distance in program memory (the principle of spatial locality). These branches typically share the same instruction format. When a given program is monitored dynamically, the skew toward offset branches becomes even more pronounced. Table 5.1 shows the branch instructions of the PISA instruction set used by HWattch (the simulator used for the experimentation). PISA is closely related to MIPS, but is designed to give results similar to other RISC instruction sets [16]. Table 5.1 also shows the static and dynamic occurrence of each branch, its associated instruction format and the possible method from the combined algorithm that can used with it.

Table 5.1: Static and dynamic branch occurrence for each PISA branch, and its occurrence across the whole EEMBC benchmark suite

Branch Instruction	Static Occurrence	Dynamic Occurrence	Branch Format (Applicable Method)
j	10.21%	17.31%	Unconditional Abs. (Local Delay Region)
jal	33.95%	3.58%	Unconditional Abs. (Local Delay Region)
jr	15.54%	3.55%	Register Jump Format (Not Hinted)
jalr	2.32%	0.04%	Register Jump Format (Not Hinted)
beq	18.18%	20.23%	Offset Format (Bias Profiling)
bne	16.46%	50.09%	Offset Format (Bias Profiling)
blez	1.52%	2.58%	Offset Format (Bias Profiling)
bgtz	0.27%	1.04%	Offset Format (Bias Profiling)
bltz	0.48%	0.39%	Offset Format (Bias Profiling)
bgez	1.06%	1.19%	Offset Format (Bias Profiling)

It can be seen that the vast majority (75%) of dynamic branch instructions are of the offset format. It is as a result of this that the decision was made to only use the profiled 'assume taken' case with this branch format. This means that the required hardware alterations (described next) can be extremely simple and require little additional decoding; with a single branch format being used, the branch target information will always be in the same bit positions within the instruction. While this seems limiting, we can also see from Table 5.1 that another major class of branch format can be covered by the local delay region. When the local delay

region is used with the unconditional absolute branch instructions, no target information is needed.

In further support of only using the 'assume taken' case with a single branch format it can be said that, even if an instruction set does not have a particular dynamic skew towards offset branches, the types of branch instructions used can be easily coerced to this end by the use of a readily available compiler techniques used to generate position independent code for secure library reuse.

5.4.2 Hardware Modifications

The use of only one branch instruction format for 'assume taken' branches means that the additional hardware logic is very simple. The modifications are implemented in two pipeline stages: instruction fetch and instruction execution. With respect to the dynamic branch predictor, this means in the prediction furnishing stage and the predictor update stage.

Instruction Fetch / Furnish Prediction

The instruction fetch stage has the more logically complex modifications of the two stages that must be altered. Out of the four statically assigned behaviours possible in a branch instruction, only two new meta-behaviours essentially need to be implemented:

1. Assume the fall-through path and do not access the predictor (local delay region or predict not-taken)

2. Use a specific bit position off-set to the current program counter and follow that target.

Alternatively, the hint-bits are both set to zero and no action is performed (the dynamic branch predictor is accessed). Figure 5.4 shows a logical representation of how these changes would be implemented in hardware.

It is difficult to state the exact hardware that would be required to implement the static predictions because it is highly implementation specific and depends on the existing predecode logic. However, the additional logic required is very limited. At most, a binary adder, a small number of logic gates and some interconnects are required. Compared to the existing control logic, the addition of these components is almost insignificant, however these are accounted for in the simulations carried out in the next chapter.

Instruction Execution / Predictor Update

The new logic required in the execution/update pipeline stage is much simpler to understand than that required in the instruction fetch stage. There are only two

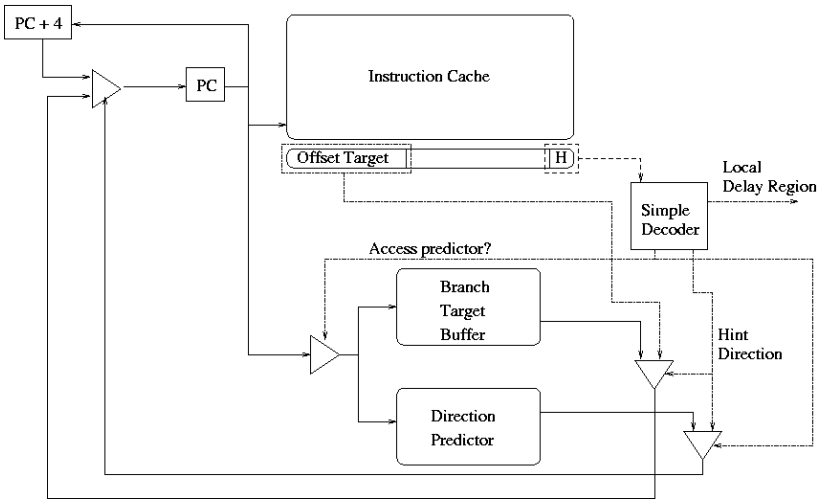

Figure 5.4: Hardware modifications required in the instruction fetch stage

modification required (represented in the previous hint-bits), and these can be easily incorporated into the existing control logic:

1. Do not update the dynamic predictor if the hint-bits are set.
2. Do not detect target misprediction for branch instructions that are hinted to use the local delay region.

The branch predictor should not be updated because, if the hint-bits are set, the updated data in the branch predictor will never be used to furnish a relevant prediction. Such an update would waste as much energy as furnishing a prediction.

The processor should ignore any prediction checking for branch instructions where the hint-bits are set to use the local delay region. In such a case, the compiler should have already scheduled the correct number of instructions into the delay slots and the calculated target of the branch instruction can be used to fetch the next instruction. In other words, the processor should not flush any instructions behind the branch in the pipeline.

These two cases can be easily incorporated into the existing hardware by using just a few gates to modify the behaviour of the current logic.

Static Branch Misprediction

It is important to note that, despite the hinted predictions for branches being static, they will still occasionally be incorrect. This does not require any special additional control logic or hardware structures (such as a reorder buffer) because there will already be a coexisting dynamic prediction system and associated recovery

mechanism. Any hinted misprediction will be handled in the same way as a hardware dynamic misprediction. The existing recovery mechanism will resolve any inconsistencies in the pipeline.

5.5 Summary

The combined approach unifies two key methods for reducing the number of accesses to a dynamic branch predictor: local delay region scheduling and profiling with an adaptive bias measurement. The selective hinting algorithm will schedule into the local delay region for unconditional absolute branches and assign static predictions to branches where profiling indicates this will not affect that branch's prediction accuracy during execution. The hinting information is reflected in two hint-bits in every branch instruction. The hardware modifications are very simple and create negligible additional energy dissipation.

The following chapters move on to experiment with the combined algorithm in a set of carefully designed experiments. The next chapter introduces the simulation tools used and created for the experiments, and discusses the specifics of how the combined algorithm is implemented in the simulation environment.

Chapter 6

Simulation Tools

6.1 Introduction

This chapter describes, in detail, the experimental method, tools and benchmarks used to generate the results for the combined algorithm, shown in the next chapter. The main tools developed are 'HWattch' (and the associated profiling and compilation tools), Hatfield Assembly Code Analyser (HACA) and a bespoke build system for Electronic Embedded Microprocessor Benchmark Consortium (EEMBC).

6.2 Simulator (HWattch)

In order to model the effectiveness of the combined algorithm presented in the previous chapter, it was necessary to choose a system of processor simulation. The two main approaches that are possible to simulate such a processor model are 'Execution-Driven' and 'Trace-Driven'.

A trace-driven simulator uses a tracing tool, such as HWattch, to generate a dynamic instruction trace of a program on a given architecture. The program trace is then stored and used as an 'unrolled execution' of a program which can then be parsed by a more simple simulator to generate different results from a fixed semantic behaviour. The main advantages of trace-driven simulation are simulator simplicity and speed of simulation; a trace-driven simulator does not need to actually execute a program, but rather just parse a fixed program trace.

In contrast, an execution-driven simulator is a program which models an actual architecture and executes code in a simulated environment. A program is compiled to the machine code of the simulator, which may or may not be the same as a 'real' architecture, and then the simulator proceeds to execute the code. The advantage of this kind of simulation is that the code execution can be modelled at various levels of detail, possibly all the way down to the gate level if the design dictates this.

For a project such as the evaluation of the combined algorithm, a trace-driven simulation would be ideal as most of the detail required would seem to be contained within the dynamic branch predictor logic. However, when measuring energy efficiency/power consumption, a more holistic view of global energy consumption is required to understand any processor-wide effects that emerge after modification of the dynamic behaviour. This can be achieved with a trace-driven simulator, but the level of simulation required is not only likely to result in only a small design simplicity improvement over an execution-driven simulator, but also makes runtime behaviour modification more difficult.

Fortunately, an existing simulation tool-chain already exists that is widely used and recognised as a reliable source for experimental results. SimpleScalar [16] is an execution-driven simulation toolkit that models various different levels of detail in different simulator executables. However, SimpleScalar does not model the energy/power consumption of a processor during execution. In 2000, a fork of the out-of-order simulator of SimpleScalar was modified to include power simulation [14] data for the entirety of the processors components. This simulator is called Wattch, and its relationship to SimpleScalar is shown in Figure 6.1. The power model was shown to be highly accurate and comparable to detailed low-level power simulation tools [14]. Wattch, as with SimpleScalar, is built in a modular fashion with each logical section of the processor broken down into a separate module. This design means that all components of the processor simulated by Wattch, from the branch predictor to issue logic, are full parameterised. This facilitates the tailoring of individual baseline models.

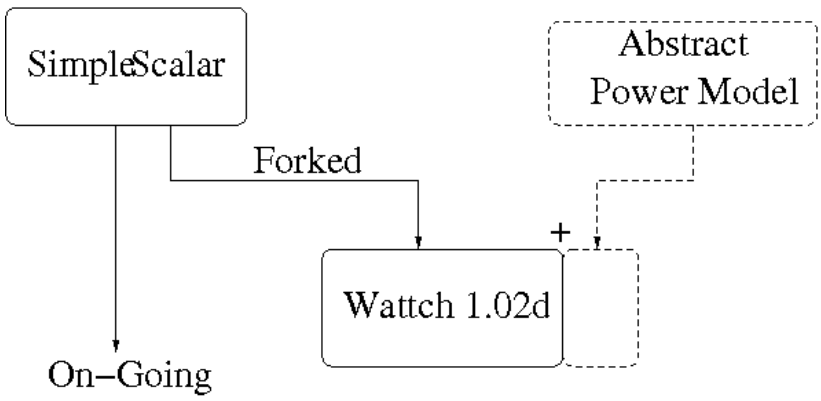

Figure 6.1: The Wattch simulator in relation to SimpleScalar

The Wattch simulator is suited to modelling the architecture necessary to analyse the combined algorithm with the EEMBC benchmarks. For this reason, Wattch was chosen as the tool to generate the results for this project. The architecture model of Wattch, and the required modifications, are explained the following subsections. *The software resulting from the modifications described in this chapter, and many other usability changes, is termed HWattch and this term will be used*

after this chapter.

6.2.1 Architecture Model

The SimpleScalar simulation toolkit provided several simulators that simulated various levels of detail and different functional units. The largest of these, called 'sim-outorder' is the union of all of the other simulators and provides the most complex results. The Wattch simulator only implements 'sim-outorder', presumably because this turned out to be the most used simulator and centralises all of the features of the toolkit in a single executable. This made it easier to insert the power model into the simulator.

'Sim-outorder' implements a full superscalar processor that uses a variant of the Tomosulo algorithm with reservation stations to schedule instructions dynamically. However, these features can be disabled and the simulator forced to behave like an inorder processor.

Pipeline

Figure 6.2 shows the general structure of the pipeline simulated by Wattch.

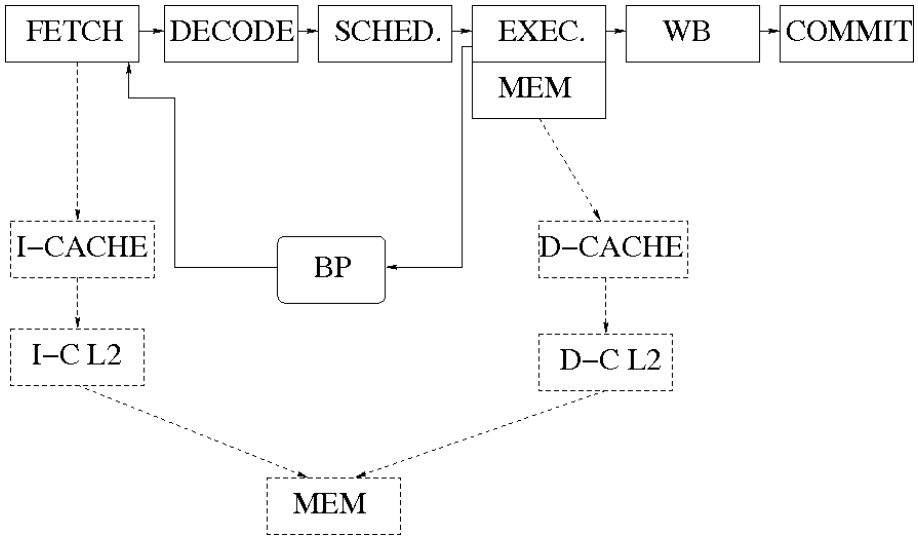

Figure 6.2: The Wattch simulator pipeline

The pipeline contains, when in scalar mode, two delay slots. When running in out-of-order mode the delay region is variable. The branch predictor is used in the Fetch and Execute stages of the pipeline to furnish a prediction and then to update the branch predictor. The cache hierarchy is broken into two levels and bifurcated into a data and instruction cache model. The register file consisters of 32 general purpose registers.

Power Model

The SimpleScalar simulator 'sim-outorder' is divided into relatively fine-grained events. For the team that designed the power model in Wattch, this was very useful [14]. The Wattch power model takes parameters for each component which specify the size (in transistors/control logic) being used in the particular simulation. From this, using a standard template for that particular unit verified from industrial gate models, Wattch creates a static power cost and a dynamic access power cost. The static power cost is a constant, per cycle, drain which is simply accumulated over the total number of cycles. The dynamic power cost models the power consumed during each transition event and is accumulated by summing these power values at the appropriate places in the SimpleScalar event methods.

The results generated from this are relative power values rather than representing any absolute incarnation. That is to say, there are atomic power values for principal events such as gate dissipation and wire capacitance which are represented as the smallest operations. Every other event in the processor is a scaled up version of these atomic interactions. The results are then consistent because the power model is consistent through the scaling relationships between operations and can be used to compare the difference between two architectures, compiler techniques and so on.

Additionally, Wattch models conditional clocking (clock gating). This is a power saving technique, described in Chapter Two, that disables the clock signal for idle processor units. For the idle unit, this can potentially have a linear power-saving effect. The clock gating models available are:

- No conditional clocking
- Simple conditional clocking (zero power dissipation with zero accesses)
- Ideal aggressive conditional clocking (linear power dissipation with fractional accesses)
- Non-ideal aggressive conditional clocking (15% power dissipation with zero accesses)

The non-ideal model is used in the results presented in the next chapter in order to give a more realistic, indeed possibly pessimistic, result.

A more detailed specification and validation of the Wattch power model can be found in the initial Wattch publication [14].

Parameters

The hardware specification of a Wattch simulation is highly configurable. The main configuration options are as follows:

IF Queue Size – The number of instructions that can be stored in the instruction fetch queue before decode/dispatch.

Prediction Latency – Although the Wattch architecture features two delay slots, an additional misprediction penalty can be added to generate results that correspond to much more deeply pipelined processors.

Front End Speed – A coefficient can be specified that Wattch will use to adjust the performance difference between the fetch stage and decode stage. For instance, one can allow x times as many instructions to be fetched as can be decoded.

Branch Prediction – The type and size of branch predictor to be used. This can include local prediction or global prediction (PAg & GAg), gshare and so on. The size is used to specify the number of entries in each table and history length in bits.

Return Address Stack – The size of the return address stack.

Branch Target Buffer – The number of entries in the branch target buffer and the set associativity.

Decode Width – The number of instructions decoded in one cycle.

Issue Width – The number of instructions issued in one cycle.

Out-of-order Execution – The processor can disable its superscalar attributes and behave as a standard scalar processor. This is only really useful when the issue width and functional units are also set to appropriate values.

Commit Width – The number of instructions committed in each cycle.

Caches – The size and speed of the caches (levels one and two) for instructions and data (separately). The associated cost of a cache miss can also be specified.

Memory Latency – The number of cycles required to access main memory.

Functional Units – The number of function units. This includes integer and floating point ALUs, multipliers and dividers.

The configuration options listed above are specified for each of the baseline experimental architectures shown in a later section.

6.2.2 Architecture Modifications

The previous chapter described the combined scheduling and profiling algorithm that the next chapter uses to generate experimental results. In that chapter, several small hardware modifications were specified so that the processor hardware can exploit the hinting information containted within specific instructions.

As with most modern embedded processors, the Wattch architecture uses pre-decode logic to determine if a dynamic instruction is a branch. This is relatively straight forward to achieve with a well designed instruction set: a small number of bits at the start of the instruction can be used to check if it is a control flow instruction. This pre-decode logic means that, already, only a branch instruction needs to access the dynamic branch predictor and will save a significant amount of power for little cost.

When this pre-decode logic exists, it means that the hint-bits discussed in the previous chapter need only be included within branch instructions. This is because when the instruction has been found to be a branch, whatever logic has just determined this can enable a logic gate which enables the previously described static prediction logic. Branch instructions invariably contain redundant bits to use for the two hint-bits required for the combined algorithm, and so this implementation is feasible. *Since Wattch implements this pre-decode logic, the hint-bits are only required in branch instructions for this study, and it is suggested that this should be the case for almost all modern architectures.*

Logic Modifications

The modifications required are:

1. Fetch Stage: If the existing pre-decode logic recognises a branch instruction then the hint-bits must be checked to see if they are set (non-zero), and if so, appropriate actions must be taken. If they are zero then nothing should happen, and the branch predictor should be accessed as normal.

2. Execution Stage: If the hint-bits are set (non-zero) then the dynamic predictor must not be updated. If the branch is found to be mispredicted, but the delay region is in use, no instruction flush or recovery should occur (as the delay region instructions must be executed and there is no misspeculation).

The above logic will ensure that power is saved and that the appropriate behaviours are modelled to generate results.

The chosen configuration of the hint-bits, detailed below, was used to simplify logic in the instruction fetch stage.

- 0 0→ Access dynamic predictor as normal
- 0 1→ Do not access the dynamic predictor. Predict not taken (leave the program counter alone)

- 1 0→ Do not access the dynamic predictor. Predict not taken (leave the program counter alone) and utilise the scheduled delay region
- 1 1→ Do not access the dynamic predictor. Predict taken by enabling an adder for a fixed mask of instruction bits (the target offset) against the current PC. Then set the PC to this value. *It is important to note that, as described in the previous chapter, only one branch format will be used to hint a taken branch. This ensures the simplicity of the logic required in the IF stage to create the target address.*

From this it can be seen that, in both the instruction fetch and execution stage, a dynamic branch predictor access/update can be blocked immediately by simply checking to see if either of the hint-bits is set. The dynamic branch predictor unit logic could be bypassed using a single OR gate whose inputs are the two hint-bits. This approach can be used in both pipeline stages. See Figures 6.3 and 6.4.

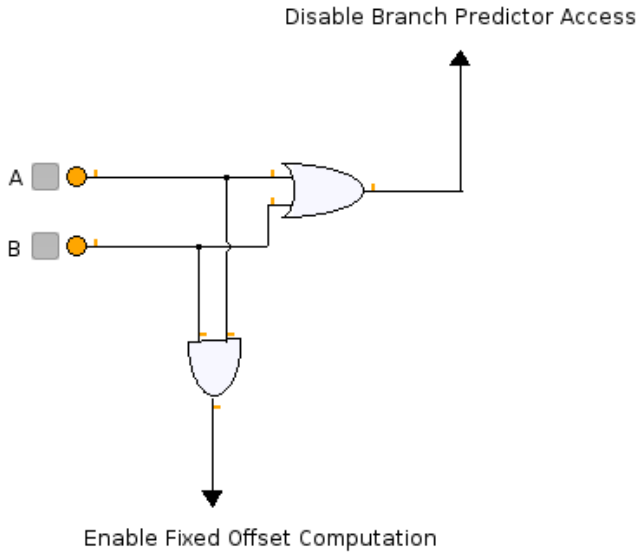

Figure 6.3: A logical represention of the IF stage hint-bits showing '1,1'

If the hints are set to '1,1' then a predict taken branch has been encountered (shown in Figure 6.3). This requires that the adder logic is enabled in the IF stage to create the new program counter value:

NEW_PC=CURRENT_PC + SIGN_EXTENDED_OFFSET.

If the hints are set to '1,0' then logic in the EXE stage must disable flushing of the pipeline (but still set the correct program counter value in the case of a taken branch).

The descriptions for the newly introduced logic are deliberately generic. This is because any implementation of the hint-bit logic is entirely dependent on design of the processor into which it is being placed. Since the aim of this project is to

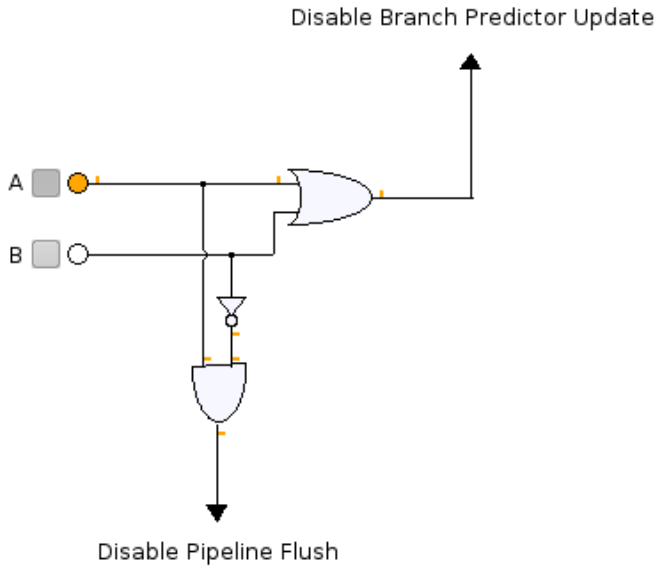

Figure 6.4: A logical represention of the EXE stage hint-bits showing '1,0'

demonstrate the general applicability of the combined algorithm, the logic descriptions have been kept as general as possible. The actual location of the hint-bits, within an instruction form, is detailed in the PISA section of this chapter.

Power Consumption of Introduced Logic

Given the aim of modelling the energy efficiency of the combined algorithm, is it prudent to make sure that any newly introduced logic is also included in the power model. From the previous subsection it can be seen that any new logic introduced into an implementation of the combined-algorithm hint-bits will be very limited. Nevertheless, the following additional logic was included into the power model (for ALL experiments):

- $1\times$ 32-bit adder
- $1\times$ 32-bit sign extender
- $1\times$ 32-bit word-line
- $8\times$ logic gates
- $20\times$ short wire capacitance

These components were created by simply replicating the power cost of their identical components containted elsewhere in the Wattch power model, and locating them in the appropriate places where the logic was introduced – the actual logic

was not explicitly introduced as Wattch is a behaviour model, and as such is written in higher level terms than logic gates.

When taken in the context of the entire processor, the introduced logic is tiny, particularly when the bulk of it will not be switching during most processor cycles, but only on a hinted taken branch;the dynamic power cost will be low even in itself.

6.2.3 Profiling Enhancement

To work correctly, the combined algorithm requires profiling information. As discussed in the previous chapter, the combined algorithm will hint a static branch direction prediction if the dynamic bias of the branch is greater than, or equal to, the dynamic branch predictor's accuracy for that branch.

The combined algorithm itself is implemented in the Hatfield Assembly Code Analyser (HACA), but this algorithm requires certain raw data to be logged during a programs execution in the form of a reduced trace. The raw information required, for a given dynamic branch occurrence, is:

1. The static identifier of the branch (used to map back the trace entry to a static assembly branch)
2. The dynamic prediction for the branch (taken or not-taken)
3. The actual dynamic behaviour of the branch (taken or not-taken)
4. Additionally, if a misprediction is detected, the number of instructions squashed. This can be used to determine the size of the delay region for a particular branch.

Before the profiling stage, HACA (detailed in 6.3) gives each static assembly branch instruction a unique identifier. This is possible by virtue of PISA (also detailed in a later section) being a 64-bit instruction set with 16 redundant bits per instruction to facilitate instruction set research (Figure 6.5). The new profiling code in Wattch then decodes this unique identifier and uses it to create the profiling entry for each dynamic branch occurrence. All profiling is performed in the 'Commit' pipeline stage.

Below is a very small excerpt from an example program trace:

```
UNIQUE ID|    BEHAVIOUR
---------------------------
...
14695         T N
22134         T T
```

```
13264          N N
...
```

Here, in line one, the predicted behaviour was taken, but the actual behaviour was not-taken. HACA then uses this trace, from an entire program's execution, to build up the required statistics for each branch.

6.2.4 Instruction Set (PISA)

The Wattch simulator, in this investigation, uses the Portable Instruction Set Architecture (PISA). PISA is a relatively simple MIPS-like instruction set that extends many of the features in Patterson and Hennessy's DLX instruction set [84] [40].

The PISA instruction set is useful because it is essentially a 32-bit instruction set that is implemented in a 64-bit form. In other words, while the PISA could be implemented in 32-bits, it was implemented as a 64-bit format so that it could include many redundant bits which can be used for instruction set research. These redundant bits can be easily set in the assembly code of a program by adding some switches to the end of an instruction. The PISA assembler then sets the appropriate bits in the machine code. A researcher can then add code into the simulator that makes use of these redundant bits. This technique was used in the development of HWattch and is detailed in the following subsections.

Instruction Format

The 64-bit PISA instruction format is shown in Figure 6.5.

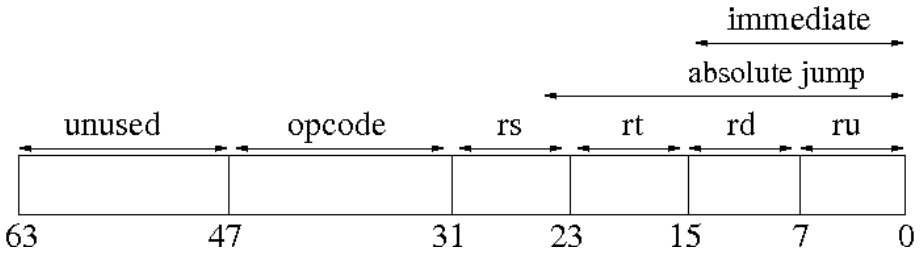

Figure 6.5: The PISA instruction format

Figure 6.5 shows three instruction formats. The bottom format is the r-type format and is the most common. The absolute format is used for branching to absolute addresses and replaces bits 0 to 26 with an absolute memory address. The immediate format replaces bits 0 to 15 with an immediate value. Each field shown has the following function:

rs – Source register one

rt – Source register two

rd – Destination register

ru – Used for shift instructions (shift amount)

opcode – Extended opcode section allowing for the introduction of new operations

immediate – In immediate type instruction, an immediate value is placed here for use by the instruction

absolute jump – Absolute jump instructions make use of a 26-bit value that is left shifted by two bits, combined with the most significant 4-bits of the current program counter, to form an effective memory address

unused – A 16-bit space is unused that allows for the introduction of new information in all instructions. These unused bits can be set using special syntax in the assembly code which is then used by the PISA assembler to enable the specified unused bits in a given instruction.

Branch Instructions

The PISA instruction set is fully featured and contains instructions for performing all required actions in a compiled program. However, the implementation of the combined algorithm is concerned only with branch instructions. The branch instructions of the PISA instruction set are shown in Table 6.1

Table 6.1: Static and dynamic branch occurrence for each PISA branch, and its occurrence across the whole EEMBC benchmark suite

Branch Instruction	Type	Static Occurrence	Dynamic Occurrence	Branch Format
j	Unconditional Abs.	10.21%	17.31%	Absolute Format
jal	Unconditional Abs.	33.95%	3.58%	Absolute Format
jr	Unconditional Reg.	15.54%	3.55%	Register Format
jalr	Unconditional Reg.	2.32%	0.04%	Register Format
beq	Cond. Offset	18.18%	20.23%	Immediate Format
bne	Cond. Offset	16.46%	50.09%	Immediate Format
blez	Cond. Offset	1.52%	2.58%	Immediate Format
bgtz	Cond. Offset	0.27%	1.04%	Immediate Format
bltz	Cond. Offset	0.48%	0.39%	Immediate Format
bgez	Cond. Offset	1.06%	1.19%	Immediate Format

From this table it can be seen that the vast majority of dynamic branches are of the immediate format (around 75% of dynamic branches). This is somewhat

intuitive, but is particularly useful for the combined algorithm which will only predict 'taken' for a single branch type.

Unconditional absolute branches (around 21% of dynamic branches)) perform no evaluation and branch immediately to the specified absolute target. Unconditional register branches (around 18% of dynamic branches) perform no evaluation and branch immediately to a value contained within a target register. Conditional offset branches (around 35% of dynamic branches) compare the values in two specified registers and then, depending on the relevant evaluation criteria, branch to the relative target offset specified within the instruction.

Hint-Bits

For the implementation of the combined algorithm in this section, two hint-bits were included as the two most significant bits in each branch instruction. This is illustrated in Figure 6.6.

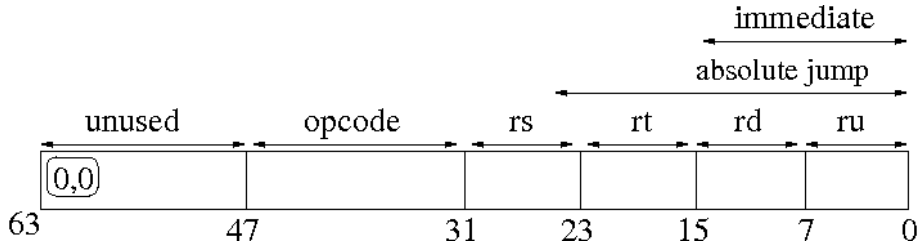

Figure 6.6: The location of the two hint-bits within the branch instruction format

The hint-bits make use of two of the redundant bits included in the PISA instruction set. Although sufficient space existed in the opcodes of branches, it was decided to use the redundant bits to simplify the implementation in Wattch. The hint-bits are stored in a consistent bit position in all branch instructions (the two highest order bits). However, two consistent bits in the branch opcode could have been used instead.

Branch Labelling

HACA assigns each static branch instruction a unique identifier, referred to as a label, so that the dynamic branch trace can be easily related back to the static branches (for use when hinting). These static identifiers are stored in the unused unused bits of the PISA instruction format. The maximum number of branches that could be hinted (in an individual executable) in this study is thus 2^{16}.

6.2.5 Compiler (Custom GCC)

Version 2.7.3 of the Gnu Compiler Chain was distributed with the SimpleScalar project to compile for the PISA instruction set. Unfortunately, an up-to-date version of this was not available, and the distributed version required many modifications for the compiler to compile and work on modern distributions of Linux. One criticism of the compiler version used is that it is quite old; stable versions of GCC are currently considered to be around 3.4, but GCC 4 has also been released. However, the benchmarks used for these experiments are all written in C code only. There have been few advances in C optimisations techniques in GCC since version 2.7.3 as most effort has been directed at increasing C++ support. This means the compiled benchmarks will still give a contemporary representation of optimised C code.

6.3 Scheduler and Static Prediction Assigner (HACA)

The Hatfield Assembly Code Analyser (HACA) is the central tool used to control the application of the combined algorithm to a code base (in this case EEMBC). The tool was written from scratch and implements various features.

HACA interfaces with HWattch and interprets the extremely long dynamic branch traces to decide how to set the static prediction hints. HACA also schedules into the local delay region where appropriate by implementing the algorithm described in Chapter 5.

6.3.1 Combined Algorithm: Practical Implementation

The most useful way to understand the features of HACA is to describe how it practically controls and applies the combined algorithm to a code base. The stages below are an approximate description of each stage in the actual application of the combined algorithm to the EEMBC instruction set. This can be compared and contrasted to the more formal description of the algorithm in the previous chapter.

1. Compile the benchmark to assembly code
2. Parse all of the assembly code and assign each static branch instruction a unique label.
3. Assemble and link the assembly code into a PISA executable
4. Using HWattch, generate the ideal (free branch prediction – the same branch predictor without any power cost) results for this benchmark using the appropriate architecture configuration
5. Using HWattch, profile the benchmark (against a ‘training’ dataset) to generate the dynamic branch trace described in the previous section

6. Statically schedule into the local delay region for unconditional absolute branches where possible, and set the appropriate hint-bits

7. Parse the dynamic branch trace and assign a static prediction where the bias of a given static branch is greater than the prediction accuracy for that branch. Set the appropriate hint-bits, but ONLY if the branch is an immediate type branch (offset branch). The introduced hardware logic can only create a target address from this kind of branch format.

8. Assemble and link the scheduled and hinted assembly code into a new PISA executable

9. Using HWattch, execute the new benchmark (against a different 'testing' dataset) and produce results for the combined algorithm version of the executable

It is important to note the use of two datasets for producing the static predictions and then testing the related power/performance change. Two datasets were used in order to reduce the possibility of overfitting to a single dataset. However, it is interesting to note that the investigation revealed that biased branches, and difficult to predict branches, tend to perform almost exactly the same, on aggregate, over all datasets within the program's domain.

When actually carried out, the above steps were performed across the entire EEMBC suite at each stage, rather than on one benchmark at a time. *The entire process takes around sixty minutes to complete on a 2.2 GigaHertz AMD Athlon64 Dual Core Gentoo Linux system with 2 GigaBytes of RAM.*

Assigning the Static Prediction

The static branch instruction bias levels are generated by creating a table of all of the static branch identifiers for that program, and then parsing the trace file for that program. As each entry in the trace file is parsed, the dynamic branch label is used to index the table of static branches and increment either the taken or not-taken counter. Additionally, the predicted behaviour for the given dynamic occurrence is compared to the actual behaviour. If these two behaviour differ then a misprediction has occurred and the misprediction counter is incremented at the appropriate index position in the table. From this information, the bias and misprediction rate can be prediction for each branch.

For a given static branch:

Branch Bias $=$ either $\frac{Dynamic\_Taken\_Total}{Total\_Dynamic\_Occurrences}$ or $\frac{Dynamic\_Not-Taken\_Total}{Total\_Dynamic\_Occurrences}$, depending on whether $Dynamic\_Taken\_Total > Dynamic\_Not-Taken\_Total$

Misprediction Rate $= \frac{Mispredictions\_Total}{Total\_Dynamic\_Occurrences}$

Assignment Condition If $Branch\_Bias > (1 - Misprediction\_Rate)$ then set the hint-bits to reflect the biased direction (can only be hinted taken for immediate (offset) format branches)

The assigning of a static prediction was only enabled for branch instructions not using the local delay region. From Table 6.1, in the previous section, this means that approximately 80% of the dynamic instruction stream can be assigned a static prediction. However, as a static predict taken can only be applied to offset type branches, approximately 75% of dynamic branches are candidates for being assigned a static prediction of taken or not-taken.

Utilising the Local Delay Region

The previous chapter described the general way to schedule into the local delay region. This technique is only used for unconditional absolute branches. These types of branch instruction have no dependency on preceding instructions as both the direction and the target are included in the instruction itself, making them ideal candidates for the local delay region scheduling. Additionally, in modern well optimised code, it is difficult to find more than one branch independent instruction from the same basic block (except for unconditional absolute branches). Ignoring other branch formats for local delay region scheduling means the static prediction through profiling can coexist more simply in the implementation.

Table 6.1, from the previous section, shows that unconditional absolute branches account for approximately 21% of the dynamic stream, and are all candidates for local delay region scheduling.

Local delay region scheduling works by filling the delay slots behind a branch instruction with branch independent instructions from the same basic block (moving instructions from before the branch into the delay region after it). This procedure takes place statically. When a delay region can be utilised, the hint-bits (discussed previously) are set to reflect this. In these experiments, the local delay region is only scheduled for unconditional absolute branches. This means that finding branch independent instructions is easy as no instructions from the same basic block are used to compute the behaviour of the branch; filling the delay region requires selecting the required number of instructions and moving them into the delay region without changing their order (so that program semantics are unaffected).

A problem arises when deciding how many instructions to fill the delay region with. On a scalar processor the delay region is of a fixed size so there is a predetermined number of slots available for instructions. However, superscalar processors use dynamic scheduling and thus have a variable delay region. Traditionally, this has meant that utilising the delay region with any form of scheduling has been difficult.

The solution used to overcome the variable delay region size is profiling. The profiling information generated by HWattch includes information about the number

of instructions squashed on a mispredicted branch. This can be used to build three useful values for each static branch: the minimum delay region slots, the mean delay region slots and the maximum delay region slots. The minimum delay slots calculated from profiling is accurate for code profiled with a dataset that 'touches' all areas of the code, however it does introduce the possibility of, under certain execution circumstances, a smaller minimum number of delay slots existing. This makes scheduling into the delay region difficult if semantics are to be maintained. Instead, a more sensible value is the theoretical minimum delay slots for the given architecture. This is determined by the number of pipeline stages before branch resolution and the instruction fetch/decode width.

To schedule into the local delay region on a superscalar processor, instructions must only be scheduled up to the minimum delay region size; if instructions were scheduled past this size it is possible that they would not be executed under certain circumstances. However, when scheduling only up to the minimum number of a delay slots, a problem still exists when considering how to stop the processor continuing to fetch instructions from the fall-through path after the delay region instructions (since the branch may be taken, and thus those instructions should not be executed). A simple solution is to insert 'NOP' instructions up to the maximum size of the delay region. This means that there could be significant code expansion (but not necessarily wasted executed instructions). The ideal solution, if the architecture supports it, is to insert a 'halt' (or similar) instruction that stops the processor from continuing to fetch instructions. The correct program counter is then used after branch resolution and the correct sequence of instructions is fetched. Execution continues. This is the method used to generate the results in this dissertation.

An alternative solution involves scheduling up to the minimum delay region size and using some additional hint-bits to mark instructions in the delay region as 'not to be flushed'. This requires additional hint-bits however, and also hardware modification to the misprediction recovery logic. The requirement of additional hardware for this technique means that it has not been investigated further in this dissertation.

Finally, even with a superscalar solution, there still exists the problem of not wasting power/clock cycles by scheduling up to the minimum delay region size when the average delay region size is much bigger than the minimum. Doing so would mean that cycles are commonly wasted since only a small number of instructions have been scheduled into a commonly much larger delay region.

Since HWattch is, potentially, a superscalar architecture, the local delay region was scheduled up to the minimum number of slots, but only if the the average size of the delay region (from profiling) was a maximum of one instruction higher. This ensures that program semantics are maintained and that a minimum number of cycles are wasted. For low issue width superscalar processors, this method is very effective.

6.4 EEMBC

EEMBC, the Embedded Microprocessor Benchmark Consortium [64], provide a suite of benchmarks that are designed to model the activities that embedded processors in a variety of different devices spend their time executing. Energy-efficient branch prediction, although relevant to the high performance arena, is of the highest important to high-end embedded processors. It is for this reason that the EEMBC suite of benchmarks was chosen for use as the evaluation benchmark for the combined algorithm. EEMBC is already used by several well known processor designers and chip manufacturers including: ARM, IBM, FreeScale, MIPS and Transmeta [64]. The suite used to generate the results in this investigation is Version 1.1, released in 2004.

6.4.1 Sub-Suites and Benchmarks

EEMBC is divided into several sub-suites: automotive, consumer, networking, office and telecom. Each of the sub-suites is designed to simulate a different kind of application's characteristics. Thirty-seven benchmarks were used in total for the results shown in this dissertation, and these are listed in full in Table 6.2.

The four fields in Table 6.2 are:

Mnemonic – The short version of the name of the benchmark. These names are used extensively in the results presented in the next chapter.

Sub-Suite – The sub-suite that each benchmark is a member of.

Relative Size – The relative dynamic size, as a percentage, of each benchmark compared to the longest. In other words, the longest benchmark (rgbcmy) is 100% and every other benchmark is measured as a percentage of this. The size of the benchmark is the number of instructions executed at runtime until completion.

Description – A short description of the main task that the benchmark carries out during execution

It can be seen that the largest benchmarks are within the consumer sub-suite which is designed to simulate the load on consumer electronics. This is expected given the high bias toward graphical algorithms that are used here; these tend to be highly iterative and run for longer periods of time. Other notably lengthy benchmarks include the networking 'qos' benchmark which uses a complex algorithm to measure the quality of service over a network hierarchy.

Although the benchmarks appear to be highly specialised, as with all benchmark suites, they tend to represent an overall whole that is not dissimilar from other benchmarking systems such as SPEC [96]. However, using these benchmarks provides an interesting insight into how the combined algorithm performs in different target applications.

Table 6.2: The full EEMBC benchmark suite with descriptions

Mnemonic	Sub-Suite	Relative Size %	Description
a2time01	Automotive	0.10	Angle-To-Time Conversion
aifftr01	Automotive	3.32	Fast Fourier Transform
aifirf01	Automotive	0.12	Finite Impulse Response(FIR)Filter
aiifft01	Automotive	3.48	Inverse Fast Fourier Transform
basefp01	Automotive	0.11	Basic Floating-Point
bitmnp01	Automotive	0.50	Bit manipulation
cacheb01	Automotive	0.19	Cache Buster
canrdr01	Automotive	0.97	Response to Remote Request
idctrn01	Automotive	0.42	Inverse Discrete Cosine Transfor
iirflt01	Automotive	0.12	Low-Pass Filter(IIR)and DSP functions
matrix01	Automotive	2.92	Matrix Math
pntrch01	Automotive	0.27	Pointer Chasing
puwmod01	Automotive	1.51	Pulse-Width Modulation
rspeed01	Automotive	0.36	Road Speed Calculation
tblook01	Automotive	0.24	Table Looku
ttsprk01	Automotive	0.69	Tooth-To-Spark
cjpeg	Consumer	13.17	JPEG Compression
djpeg	Consumer	89.26	JPEG Decompression
rgbcmy	Consumer	**100.00**	RGB to CMY
rgbhpg	Consumer	10.73	Grayscale image filter
rgbyiq	Consumer	83.00	RGB to YIQ
ip_pktcheck	Network	10.70	IP Packet Check
ip_reassembl	Network	13.42	IP Reassembly
nat	Network	14.31	Network Address Translation
ospfv2	Network	1.90	Open Shortest Path First
qos	Network	46.33	Quality of Service
routelookup	Network	5.07	Route Lookup
tcp	Network	0.24	TCP-BM
bezier01	Office	2.83	Bezier Curve Interpolation
dither01	Office	17.13	Floyd-Stein Grayscale Dithering
rotate01	Office	4.91	Bitmap Rotation
text01	Office	11.09	Text Parsing
autcor00	Telecom	0.35	Autocorrelation
conven00	Telecom	0.81	Convolutional Encoder
fbital00	Telecom	1.00	Bit Allocation
fft00	Telecom	0.28	FFT/IFFT
viterb00	Telecom	0.63	Viterbi Decoder

6.4.2 Bespoke Build System for the Combined Algorithm

EEMBC is distributed with its own cross-platform build system. However, this system does not allow the user to compile the benchmarks down to assembly code, and then from the assembly code into binary form. This is a problem because the combined algorithm, implemented in HACA, requires access to the assembly code of the benchmarks in order to modify them and apply hinting/scheduling techniques. The assembly code then needs to be built into binary form with this hinting and scheduling information contained within it.

A bespoke build system was developed for the EEMBC suite to this end. This was a non-trivial task that required an intimate understanding and interpretation of the existing Makefiles for EEMBC. Additionally, any internal shared source files within EEMBC were copied into each individual benchmark that used them, to enable non-conflicting hinting of shared source code.

6.5 Summary

This chapter has introduced the experimental toolkit that will be used to evaluate the effectiveness of the combined algorithm. The central tool is the parameterisable superscalar processor simulator 'HWattch'. HWattch has a five stage MIPS-like pipeline, but has a variable branch misprediction penalty (set via a runtime parameter). The HWattch simulator is essentially the Wattch power-aware processor simulator (itself a version of SimpleScalar), but has been modified to include some simple control logic that interprets two new hint-bits contained within branch instructions. This logic can bypass the dynamic predictor for specified branches, and also produce a target target address for a certain class of branches.

There are two hint-bits inside each branch instruction. These hint-bits, when set by the compiler for a particular branch, allow for the use of a locally scheduled delay region, to assume taken, to assume not taken or, finally, use the dynamic branch predictor as normal.

The HACA tool is used to profile and set these hint-bits in the compiled assembly code for those branches where either the local delay region can be scheduled, or where a static prediction is possible (when the branch is more biased than its profile dynamic prediction).

The EEMBC benchmark suite was chosen as the target application area is the embedded, and hence power, sensitive market. The EEMBC suite represents the most relevant cross-section of benchmark activities likely to be carried out in such environments.

The next chapter uses the all of the tools discussed in this chapter to test the combined algorithm against several baseline configurations of the HWattch simulator. These baselines are based on example embedded configurations, and the results are presented and discussed separately for each baseline model.

Chapter 7

Simulations and Results

7.1 Introduction

This chapter presents the main body of simulation results generated by using the combined algorithm to reduce dynamic branch predictor accesses in the EEMBC benchmarks. All of the simulation results were generated using the tools and methods described in the previous chapter. Several baseline models are discussed and described in detail before the results are presented in the following three sections. Each result set is followed by a brief discussion, and then the chapter closes with a final analysis of the results, and the questions arising from them.

7.2 The Baseline Models

When assessing the effectiveness of an alteration to a compilation process and processor architecture, it is necessary to establish some model of comparison. With the evaluation of the combined algorithm, the assessment criteria are: energy/power saving, change in execution time and the reduction in the number of accesses made to the dynamic predictor. However, any results generated are indicative only for the given baseline model. To give a more varied context to the combined algorithm, several baseline models have been chosen. These are based around a number of real embedded architectures and are described in detail on the following pages. Although the baseline models are based around real architectures, it must be noted that this is only achieved by the adjustment of the available parameters in HWattch. Not all aspects of the architectures can be completely represented here, but the baselines give an appropriate indication. Each baseline model was used to generate a separate set of results.

7.2.1 The Branch Predictor

The dynamic branch predictor type used in these experiments is a local predictor. Although the combined algorithm essentially treats the dynamic branch predictor as a black box, some discussion and justification of this predictor type is required.

The efficacy of the directional hinting in the combined algorithm will likely be closely linked to the accuracy of the dynamic branch predictor it is used with; the less accurate the branch predictor, the greater the number of branches with a bias higher than the predictors accuracy. The converse is also true. To achieve a suitably rigorous and suitably sceptical result, it is sensible to use the most accurate predictor available; a large local branch predictor will saturate at 97.1% correct. This is higher than other common predictor configurations.

The traditional argument against local prediction, despite its accuracy, is that it often requires large history tables to achieve this accuracy and also, originally, requires two sequential table lookups (see chapter two). The rebuttal to these arguments is that the additional size required for local prediction is no longer a particular concern given the increased number of transistors available in modern processors. Sequential table lookups are undeniably slower, but this is not the problem it seems as a fast implementation can be achieved by using a separate bimodal counter array for each instruction fetched, so that the second array access can proceed in parallel with instruction fetch. Additionally, this is likely to not even be necessary as the clock speeds in embedded processors are considerably lower than the high performance market (allowing plenty of time for two sequential lookups).

A salient alternative choice of dynamic branch predictor type would be a gshare predictor. This implements a variant of global prediction. Large gshare predictors are not quite as accurate as their local counter parts, but table lookups can be easily made in parallel. Another noteworthy point about a gshare predictor is that, due to it being a global predictor, it may be expected that the combined algorithm by removing certain predictor updates, could affect the accuracy of the dynamic predictor on the remaining branches by witholding required path information. This was not noticed to any significant extent in our previous simulations [44].

If a global prediction paradigm had been chosen, and experimentation showed prediction problems, the hardware could be modified to allow the updating of a history register. The history register is a very small part of the direction predictor, and would have minimal impact on the efficacy of the combined algorithm's power saving. Updating the history register removes the impact on the prediction accuracy of a global predictor.

7.2.2 Scalar Processor

The scalar processor baseline is designed to give HWattch the behaviours and costs associated with a common contemporary embedded processor. By far the most common embedded processor designs currently in use are variants of the ARM architecture. ARM processors are found in almost all varieties of electronic devices,

particularly in consumer electronics such as PDAs, mobile telephones and media players. The configuration described here is shown in Table 7.1.

There is a large variety of ARM cores, and it is difficult to match exactly the architecture of any in particular. However, the general functional units and the size of caches can be modelled. This will give a strong indication of how effective the combined algorithm will be for such existing embedded architectures.

The specific architectures on which this baseline is based are the: ARM9E, ARM10E, ARM11 and embedded PowerPC implementations. All of these are scalar uniprocessors (with the exception of the ARM11 which can be used in a multicore configuration). The number of pipeline stages varies from three to eight. All have relatively small caches compared to the performance market. The baseline in Table 7.1 most closely models the ARM11 architecture.

It should be noted that HWattch, being essentially Wattch/SimpleScalar, models an out-of-order issue superscalar architecture. The parameters specified here then force that architecture to behave as in-order and single issue. In terms of simulated behaviour, this will behave exactly as an in-order processor would be expected to. However, the Wattch power metrics take into account certain structures whose simulated power consumption is not completely affected by the forced specification of an in-order processor. This means that the Wattch power model still models the power consumption of scheduling logic even though this logic is never used since the processor is scalar. The effects of this are discussed in the results analysis.

7.2.3 Multiple Instruction Issue Processor

The vast majority of embedded processors are scalar by design. Superscalar processors would appear, in theory, to offer a more efficient utilisation of resources by issuing instructions into each kind of functional unit in parallel. However, the scheduling and instruction issue logic required to achieve this has traditionally consumed too much power to make superscalar processors a viable option.

Contemporary fabrication technologies have, as discussed in chapter two, reduced the feature size of logic components used to construct CMOS circuits. With this, the overhead, in terms of power, of introducing new logic into a processor has decreased. Many new embedded architectures are proposing Multiple Instruction Issue (MII) techniques to achieve a higher performance. Typically, these take the form of Very Long Instruction Word (VLIW) architecture, but some designs use dynamic scheduling techniques too.

The HWattch simulator models a superscalar architecture, which means that this study utilises dynamic scheduling. Although results generated from this method will not be directly applicable to VLIW processors, they will show useful results since the logical behaviour of the branch predictor in the instruction fetch stage remains the same. Proportionally, this will still give a useful indication of the throughput effects of the combined algorithm, and how much power can be saved

Table 7.1: Scalar Processor Baseline Configuration

HWattch Parameter	Value
Instruction Fetch Size	1
Delay Slots/Mispredict Penalty	2
Branch Predictor	2-Level Local - 1024 Table Entries
Branch Target Buffer Size	512 Entries, 4-Way Set Associative
Average Global Power Consumption of BP	9.9%
Decode Width	1
Out-of-order Issue	No
Commit Width	1
L1 Data Cache Size	32 Entry, each entry 2-Way Set Associative
L1 Instruction Cache Size	32 Entry, Direct Mapped
L1 Hit Latency	1 Cycle
L2 Data Cache Size	64 Entry, each entry 2-Way Set Associative
L2 Instruction Cache Size	64 Entry, each entry 2-Way Set Associative
L2 Latency	6
Memory Latency	18 Cycles for first block, 2 thereafter
Integer ALUs	1
Integer Multipliers/Dividers	1
Floating Point ALUs	1
Floating Point Multipliers/Dividers	1
Clock Gating Regime	Non-Ideal

globally with its inclusion.

It is important to note that the delay region in these baseline models is now variable. Consequently, the local delay region scheduler now employs the technique for scheduling a variable size delay region.

Two Instruction Issue

The two instruction issue baseline model is intended to represent the next generation of embedded high performance processors. It is a modified version of the scalar baseline model with a two instruction issue width and dynamic instruction scheduling onto an appropriate number of functional units. This architecture is shown in Table 7.2. Some current architectures, such as certain implementation of the ARM11, already have multiple functional units and a multiple instruction issue width; this baseline model can be considered similar to these architectures.

Sixteen Instruction Issue

It is possible that the combined algorithm could reduce the prediction accuracy of branches that are assigned a static 'hint' prediction. Although it is currently uncommon to see any embedded processors with an issue width of sixteen instructions, it is interesting to see how the combined algorithm changes the behaviour of such high issue widths. It is for this reason that the architecture shown in Table 7.3 is used to generate results.

7.3 Preamble To Results

The previous two chapters have discussed both the abstract concept of the proposed combined algorithm, and also the specific experimental implementation and method. During these chapters, a great deal of information has been covered. This section discusses the appropriate metrics used to display the experimental results, and brings forward some important details that should be borne in mind before examining the results.

7.3.1 Metrics

The combined scheduling, profiling and hinting algorithm could potentially affect: dynamic branch predictor accesses, dynamic prediction accuracy and power consumption. To facilitate the monitoring of this behaviour, the following metrics are used in the results:

Benchmark – The mnemonic for the particular benchmark.

Table 7.2: 2-Way Issue Processor Baseline Configuration

HWattch Parameter	Value
Instruction Fetch Size	2
Delay Region/Mispredict Penalty	Variable (Average = 2)
Branch Predictor	2-Level Local - 1024 Table Entries
Branch Target Buffer Size	512 Entries, 4-Way Set Associative
Average Global Power Consumption of BP	8.8%
Decode Width	2
Out-of-order Issue	Yes
Commit Width	2
Register Update Unit Size	8
Load/Store Queue Size	4
L1 Data Cache Size	32 Entry, each entry 4-Way Set Associative
L1 Instruction Cache Size	32 Entry, Direct Mapped
L1 Hit Latency	1 Cycle
L2 Data Cache Size	64 Entry, each entry 4-Way Set Associative
L2 Instruction Cache Size	64 Entry, each entry 4-Way Set Associative
L2 Latency	6
Memory Latency	18 Cycles for first block, 2 thereafter
Integer ALUs	2
Integer Multipliers/Dividers	1
Floating Point ALUs	2
Floating Point Multipliers/Dividers	1
Clock Gating Regime	Non-Ideal

Table 7.3: 16-Way Issue Processor Baseline Configuration

HWattch Parameter	Value
Instruction Fetch Size	16
Delay Region/Mispredict Penalty	Variable (Average = 8)
Branch Predictor	2-Level Local - 1024 Table Entries
Branch Target Buffer Size	512 Entries, 4-Way Set Associative
Average Global Power Consumption of BP	6.7%
Decode Width	16
Out-of-order Issue	Yes
Commit Width	16
Register Update Unit Size	16
Load/Store Queue Size	16
L1 Data Cache Size	32 Entry, each entry 4-Way Set Associative
L1 Instruction Cache Size	32 Entry, Direct Mapped
L1 Hit Latency	1 Cycle
L2 Data Cache Size	64 Entry, each entry 4-Way Set Associative
L2 Instruction Cache Size	64 Entry, each entry 4-Way Set Associative
L2 Latency	6
Memory Latency	18 Cycles for first block, 2 thereafter
Integer ALUs	8
Integer Multipliers/Dividers	8
Floating Point ALUs	8
Floating Point Multipliers/Dividers	8
Clock Gating Regime	Non-Ideal

Dynamic Branches – The percentage of the *dynamic* stream accounted for by branch instructions.

Hint Rate – The percentage of *static* branch instructions that are hinted to use either the local delay region or a static branch prediction.

Access Reduction – The percentage reduction in accesses to the dynamic branch predictor as a result of the hint-bits inserted into branch instructions by the combined algorithm. This metric is the central goal of the combined algorithm. It can be regarded as the central representative of the activity factor in the general equation for power saving. Reducing the number of accesses here will reduce the activity factor.

Stream Change – The percentage change in the number of instructions executed during a given benchmark's execution. This is affected by any change in prediction accuracy caused by the combined algorithm. The stream change percentage can be either positive (the number of instructions executed has increased) or negative (the number of instructions executed has decreased). A positive change indicates a relative reduction in branch prediction accuracy over the baseline, but a decrease indicates the converse.

Cycles Change – The percentage change in the number of cycles during a given benchmark's execution. The implication of this change is almost identical to the previous metric.

BP Power Saving – The percentage saving, in terms of power, within the branch prediction unit. It would be logical for a reduction in accesses, and thus switching activity, to reduce the amount of power consumed by the branch prediction unit.

Global Power Saving (Central Power Metric)

The main metric used to model the effects of the combined algorithm on power consumption, and energy efficiency, is *"Average Percentage Power Saving Per Committed Instruction"*. This metric is labelled as simply 'Power Saving" in the results.

Deciding upon an appropriate metric for global power saving is difficult. It is easy to measure the power saved within the dynamic branch predictor unit, but this does not give an indication of how much power has been saved globally. Additionally, it does not take into account any power consumed by any change in the duration of program execution; if the combined algorithm decreases branch prediction accuracy, more instructions will be executed and global power consumption could increase.

"Average Percentage Power Saving Per Committed Instruction" is calculated as follows:

Power Saving= $\frac{Total\_Processor\_Power\_Reduction}{Number\_of\_Committed\_Instructions}$

Where "Total Processor Power" is the total power consumed by the processor over the benchmark's entire execution.

This metric gives a direct representation of global power saving resulting from the use of the combined algorithm. Additionally, it implicitly takes into account any change in branch prediction accuracy; if overall prediction accuracy is decreased, the power saving will decrease proportionally.

The power saving metric used here was chosen over the 'et^2' metric because the 'et^2' metric is mostly useful for drawing an abstraction over the power consumption of an actual implementation of hardware so that it can be compared to other implementations. This is not necessary in the case of HWattch as the constituent power metrics are already completely abstract. Additionally, it is argued that 'et^2' gives an unfair, quadratic, bias to any change in execution time of a given hardware modification [46].

Free/Ideal Branch Prediction

In many of the graphs in the results section, a metric called 'Free/Ideal' branch prediction is shown. This metric represents the ceiling power saving that could be achieved with the combined algorithm. Free branch prediction is the baseline dynamic predictor, but with zero power cost. It was debated whether or not to show perfect branch prediction with zero power consumption instead, but giving a value for perfect prediction would be too unrealistic for an algorithm that aims only to affect the power consumption of a branch predictor.

7.3.2 Calculation of Averages and 'Weighted Averages'

Each section of results consists of a breakdown for each benchmark, but also the average values for each sub-suite. These averages are not just a simple mean of all of the results, but rather they take into account the length/time, in executed instructions, that each benchmark within a sub-suite lasts. This is achieved by totalling and averaging the relevant values across the entire subsuite. For instance, when calculating the average dynamic predictor access reduction for a sub-suite, the *total* number of dynamic accesses removed for that sub-suite is calculated, and then divided by the *total* number of access made for the sub-suite without the algorithm. This gives an accurate average for the relative size, in execution time, of each sub-suite. Hence, the term 'weighted average' is used to reflect the fact that these average values are not a crude mean. The raw data from which all of the average values were calculated, along with an explanation of their presentation, is available in Appendix D.

7.3.3 Important Summary Notes

- All benchmarks shown in the following results were executed to completion.
- The combined algorithm can potentially work on a total of around 82% of static branches. This is because the hint-bits are only set where the hardware logic can handle a statically predicted behaviour (as discussed in the previous chapter). Dynamically, the percentage of potential branches is around 96% (around 18% by local delay region scheduling).
- All benchmarks were profiled individually.
- All results are in percentages.
- Details of the breakdown of the use of the local delay region versus static hinting are not shown because, in the first two baseline models, the utilisation of the local delay region is almost consistently 100% of applicable branch types and the average amount of branches for which it is applicable remains almost constant at around 17%.
- All of the raw data from which the averages and percentages were calculated is available in Appendix D.

7.4 Scalar Processor Results

The results in this section demonstrate the effects of using the combined algorithm on the EEMBC suite and scalar processor baseline.

7.4.1 Benchmark Breakdown

Table 7.4 shows the resulting change in the described metrics for each benchmark on the scalar baseline processor after applying the combined algorithm to the entire EEMBC suite. The graph in Figure 7.1 shows the global power saving metric compared with free/ideal branch prediction.

It can be seen from Table 7.4 that the most successful results, in terms of power saving, are in benchmarks: canrdr01, matrix01, pntrch01, puwmod01, rspeed01, ttsprk01, rgbcmy, ip_pktcheck, ospfv2, routelookup and rotate01; global processor power savings are as high as 13.34%. These benchmarks are also generally associated with the highest dynamic predictor access reductions – as high as 85.78%. Additionally, these benchmarks are also associated with a significant decrease in the number of instructions executed and the number of cycles that the simulation lasted. Some benchmarks have a very slight increase in the number of instructions/cycles executed. The hint rate for the successful benchmarks does not correlate with the power saving efficacy. The 'successful' benchmarks are spread, in number, relatively equally across all of the EEMBC sub-suites. The degree

Table 7.4: Benchmark breakdown results for scalar baseline processor

All results are percentages

Benchmark	Dynamic Branches	Hint Rate	Access Reduction	Stream Change	Cycles Change	BP Power Saving	Global Saving
a2time01	19.45	27.03	9.43	1.42	1.58	12.66	1.32
aifftr01	12.59	28.08	61.26	-0.52	-0.17	58.49	3.36
aifirf01	15.79	26.53	23.15	-2.10	-1.77	25.13	2.06
aiifft01	12.61	26.42	57.76	4.80	4.30	56.32	3.37
basefp01	18.05	22.94	24.66	-0.04	0.02	28.75	2.94
bitmnp01	16.00	19.83	35.46	0.11	-0.07	34.52	2.52
cacheb01	14.22	18.18	56.26	-0.54	-0.18	55.46	3.42
canrdr01	22.79	19.41	69.27	-1.23	-0.51	66.11	9.85
idctrn01	12.02	16.25	37.72	-0.24	-0.29	39.79	3.46
iirflt01	16.47	37.08	15.38	-0.04	-0.05	17.98	1.86
matrix01	18.14	17.82	54.52	-8.52	-9.28	52.34	6.62
pntrch01	22.61	26.87	63.27	-0.80	-0.53	62.20	8.02
puwmod01	22.96	14.12	70.19	-1.26	-0.49	66.83	10.05
rspeed01	22.70	22.02	67.64	-4.02	-6.65	62.39	11.51
tblook01	21.43	30.11	20.03	0.64	1.11	23.00	2.51
ttsprk01	22.67	11.80	67.65	-1.10	-0.16	64.62	9.10
cjpeg	10.40	30.84	85.78	0.00	-0.12	87.75	5.93
djpeg	14.43	32.34	49.44	0.00	-0.04	52.20	4.65
rgbcmy	22.22	24.84	69.68	0.44	0.13	66.43	8.77
rgbhpg	12.78	20.00	74.78	-0.55	-0.21	72.23	4.74
rgbyiq	15.56	23.38	31.55	0.00	0.00	33.83	2.90
ip_pktcheck	22.54	6.20	70.28	0.35	-0.06	67.48	9.38
ip_reassembly	20.18	12.34	65.83	0.27	-0.04	62.21	5.62
nat	23.26	12.84	68.02	0.31	0.03	64.46	7.76
ospfv2	28.51	6.29	83.32	0.23	0.00	80.72	13.34
qos	20.92	12.67	35.29	-0.05	-0.01	36.29	3.93
routelookup	23.73	6.03	69.58	-1.05	-0.34	67.07	10.17
tcp	14.29	12.20	10.55	-0.77	-0.94	9.25	0.70
bezier01	10.95	17.93	57.49	-0.46	-0.17	53.42	2.46
dither01	16.12	18.54	28.93	0.00	-0.02	31.93	3.31
rotate01	25.69	44.29	66.32	-0.28	-1.28	63.49	9.75
text01	24.31	33.24	46.65	-0.40	-0.70	42.40	5.67
autcor00	9.21	20.28	65.69	-0.10	-0.12	68.34	4.79
conven00	16.21	18.75	25.45	0.00	0.00	25.87	2.07
fbital00	15.06	20.65	50.19	0.02	0.06	51.33	5.50
fft00	12.10	22.94	46.55	0.14	0.01	44.86	2.57
viterb00	9.49	18.24	49.45	-0.01	-0.01	58.35	3.55

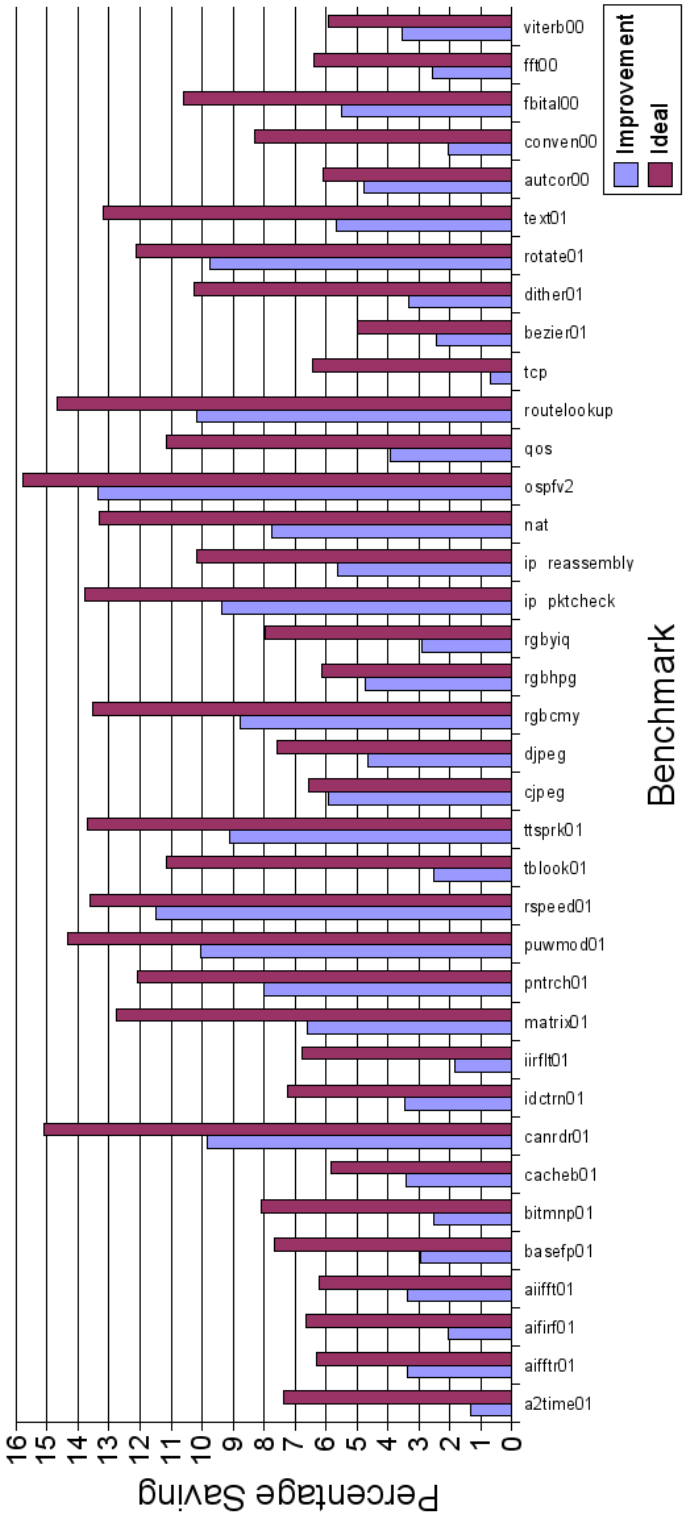

Figure 7.1: Scalar baseline global power savings (%) compared with ideal (free) prediction

to which the ideal power saving is achieved varies greatly between benchmarks, proportionally to the reduction in dynamic predictor accesses. Importantly, every benchmark registers a power saving.

7.4.2 Averages

Table 7.5 shows the weighted averages of the previous scalar results for each of the EEMBC sub-suites. The graph in Figure 7.2 shows the same scalar averages compared with free/ideal branch prediction.

Table 7.5: Average benchmark results for scalar baseline processor
All results are percentages

Benchmark	Dynamic Branches	Hint Rate	Access Reduction	Stream Change	Cycles Change	Global Saving
Automotive	15.88	22.08	59.86	-0.44	-0.28	5.04
Consumer	16.19	30.65	61.17	0.1	-0.01	5.93
Network	21.97	10.57	58.18	0.08	-0.04	6.53
Office	18.76	30.88	45.62	-0.22	-0.45	4.88
Telecom	12.11	20.23	47.84	-0.01	-0.02	4.07
Overall	17.20	18.25	59.51	0.06	-0.05	5.92

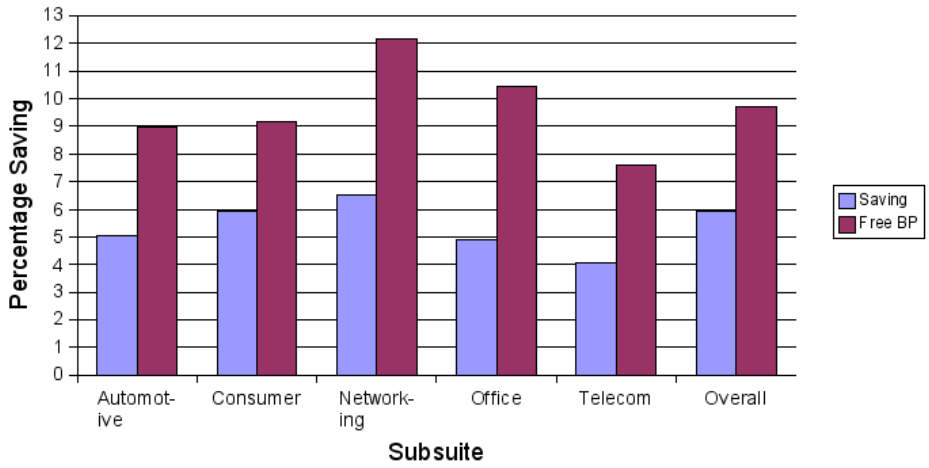

Figure 7.2: Scalar baseline average global power savings (%) compared with ideal (free) prediction

The sub-suite averages show that the combined algorithm is, overall, consistently effective in terms of power saving. The most positively affected sub-suite is ‘networking’, however this is also the sub-suite that consumes the most branch prediction power. The sub-suite that most closely achieved its ideal is ‘consumer’. Overall, the power savings were closely matched, at 6%, between the sub-suites,

and missed the ideal by around 3.5%. In no case is there a power increase. The use of the local delay region is near 100% for those branches where it is applicable.

7.5 Two Instruction Issue Processor Results

The results in this section demonstrate the effects of using the combined algorithm on the EEMBC suite and two-way instruction issue processor baseline.

7.5.1 Benchmark Breakdown

Table 7.6 shows the resulting change in the described metrics for each benchmark on the scalar baseline processor after applying the combined algorithm to the entire EEMBC suite. The graph in Figure 7.3 shows the global power saving metric compared with free/ideal branch prediction.

In Table 7.6 it can be seen that the most successful benchmarks are similar to those in the scalar baseline results. The best performing benchmarks, in terms of global power saving, are: canrdr01, matrix01, pntrch01, puwmod01, rspeed01, ttsprk01, rgbcmy, ip_pktcheck, ospfv2, routelookup and rotate01. The highest power saving is lower this time though: 12.42%. This is generally the same with all of the power saving values. Internal power saving in the branch predictor remains similar, but power savings generally come much closer to the ideal values of free branch prediction. The hint rate of static branch instructions is almost exactly the same. An important shift in the results for this MII processor is that the change in the number of instructions/cycles executed is now more pronounced, and almost all benchmarks are showing a decrease in the number of instructions/cycles executed.

7.5.2 Averages

Table 7.7 shows the weighted averages of the previous two-way issue results for each of the EEMBC sub-suites. The graph in Figure 7.4 shows the same two-way issue averages compared with free/ideal branch prediction.

The most salient fact from the average results is that the combined algorithm's power saving is much closer to the ideal of free branch prediction on this baseline model. Overall there is only around a 2.5% difference between the actual saving of 6.25% and the ideal of 8.6%. The change in the number of instructions/executed is more positive on this baseline model, with all sub-suites showing fewer cycles during execution. The efficacy of the local delay region scheduling is not shown in the averages, but remains close to 100% for those branches where it is applicable.

Table 7.6: Benchmark breakdown results for two-way issue baseline processor

All results are percentages

Benchmark	Dynamic Branches	Hint Rate	Access Reduction	Stream Change	Cycles Change	BP Power Saving	Global Saving
a2time01	19.88	27.03	9.12	1.03	0.76	11.92	1.58
aifftr01	13.64	28.08	63.14	1.05	0.03	59.49	3.72
aifirf01	16.22	26.53	20.16	-0.12	-0.06	24.37	1.79
aiifft01	13.70	26.42	64.30	0.20	-0.95	60.22	3.73
basefp01	18.48	22.94	22.89	0.11	0.16	28.19	2.10
bitmnp01	16.38	19.83	38.13	-0.29	-0.86	35.88	2.68
cacheb01	15.31	18.18	52.45	-0.32	-0.93	53.45	3.88
canrdr01	23.92	19.41	64.02	-1.56	-4.89	62.30	10.82
idctrn01	12.27	16.25	36.86	-0.35	-0.60	39.24	4.10
iirflt01	16.74	37.08	15.21	-1.60	-1.44	17.67	1.64
matrix01	19.08	17.82	50.79	-0.01	-0.73	53.43	6.13
pntrch01	23.33	26.87	59.62	0.13	-0.68	60.35	7.40
puwmod01	24.09	14.12	64.44	-1.02	-3.70	63.07	10.44
rspeed01	23.78	22.02	59.77	-0.39	-1.69	59.41	8.66
tblook01	22.42	30.11	24.17	-0.61	-0.75	24.11	3.29
ttsprk01	23.79	11.80	61.79	-0.35	-1.72	61.24	8.97
cjpeg	11.05	31.09	86.30	-0.22	-0.77	87.97	5.60
djpeg	15.23	32.44	51.40	-0.34	-0.86	52.63	4.78
rgbcmy	23.22	24.84	72.87	0.62	-1.50	68.09	9.13
rgbhpg	13.79	20.00	69.23	-0.13	-0.87	69.38	4.97
rgbyiq	16.69	23.38	36.46	-0.49	-1.16	35.72	3.79
ip_pktcheck	23.39	6.20	69.08	1.22	-0.82	64.11	8.50
ip_reassembly	21.10	12.25	69.41	0.94	-1.03	64.03	6.34
nat	23.87	12.88	71.47	0.43	-1.67	66.19	8.29
ospfv2	28.41	6.29	83.69	-2.43	-4.22	80.37	12.42
qos	23.36	12.72	55.84	-0.02	-0.08	53.69	5.91
routelookup	24.78	5.93	59.32	-0.30	-1.70	58.53	8.66
tcp	14.78	12.13	11.04	-0.66	-0.89	9.30	0.92
bezier01	12.16	17.93	56.14	-0.19	-0.79	53.06	3.18
dither01	16.54	18.54	29.69	-0.16	-0.36	32.33	3.17
rotate01	27.08	44.29	70.28	-1.22	-3.77	63.84	11.04
text01	24.84	33.24	46.19	-1.20	-3.51	40.51	6.62
autcor00	9.36	20.28	47.17	-0.12	-0.21	49.05	3.08
conven00	16.83	18.75	28.47	-0.47	-0.79	27.28	2.51
fbital00	15.19	20.65	49.96	-0.14	-0.21	51.20	4.81
fft00	12.72	22.94	39.88	-0.02	-0.57	36.56	2.49
viterb00	9.52	18.24	50.61	-0.13	-0.77	58.62	3.19

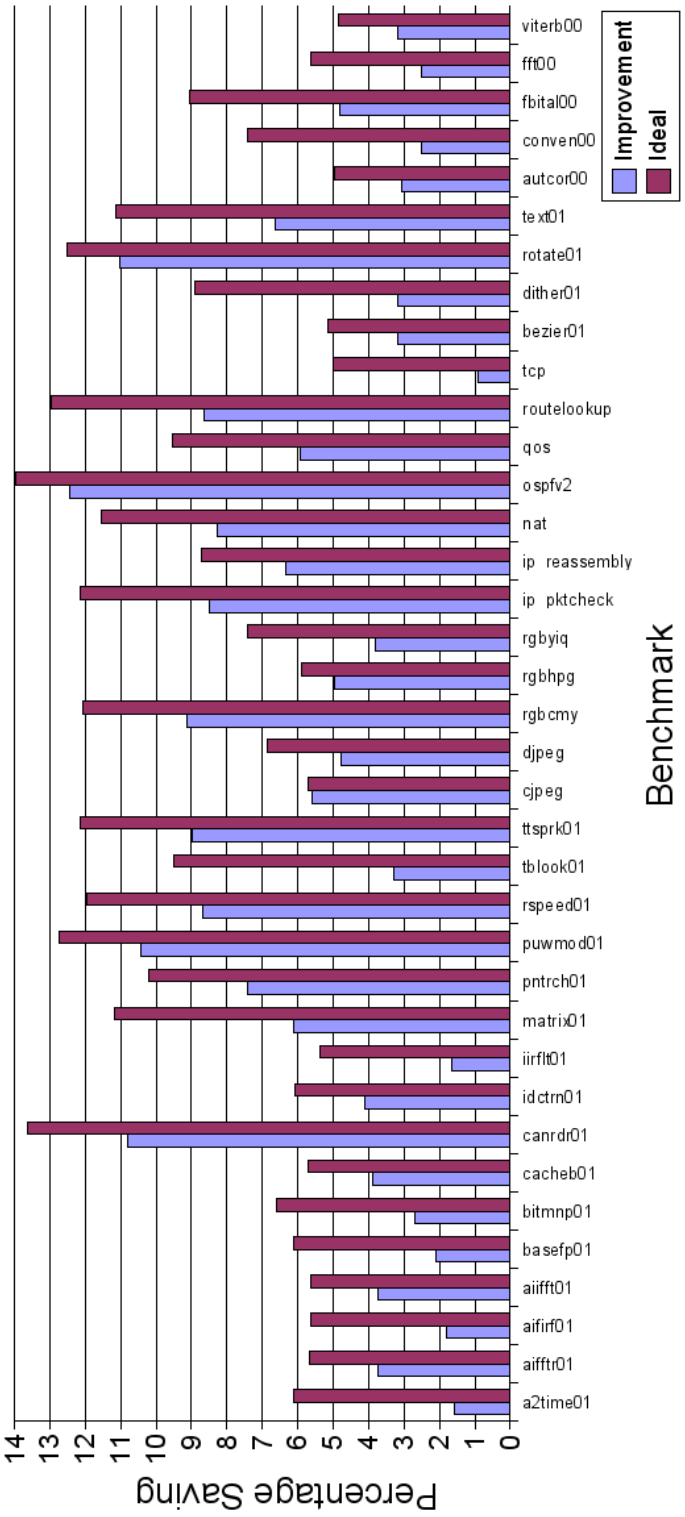

Figure 7.3: 2-way issue baseline global power savings (%) compared with ideal (free) prediction

Table 7.7: Average benchmark results for two-way issue baseline processor
All results are percentages

Benchmark	Dynamic Branches	Hint Rate	Access Reduction	Stream Change	Cycles Change	Global Saving
Automotive	16.95	22.08	59.61	0.11	-1.14	5.28
Consumer	17.23	30.80	63.56	-0.01	-1.1	6.17
Network	23.32	10.56	65.07	0.34	-0.89	7.19
Office	19.61	30.88	47.07	-0.6	-1.84	5.37
Telecom	12.11	20.23	44.62	-0.16	-0.45	3.39
Overall	18.29	18.28	62.64	0.01	-1.1	6.25

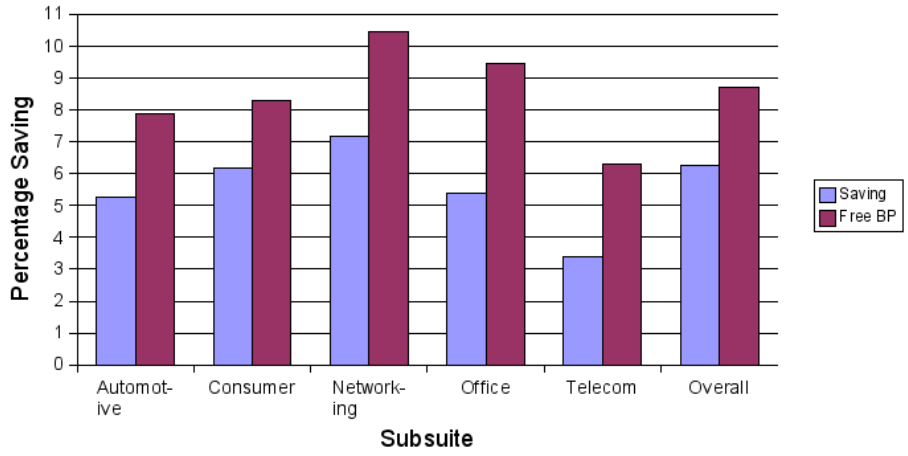

Figure 7.4: 2-way issue baseline average power savings (%) compared with ideal (free) prediction

7.6 Sixteen Instruction Issue Processor Results

The results in this section demonstrate the effects of using the combined algorithm on the EEMBC suite and sixteen-way instruction issue processor baseline.

Position Independent Code

The combined algorithm uses local delay region scheduling for unconditional absolute branches. Unfortunately, in superscalar processors, this can only be used with very small issue widths. This is because of the difficulty associated with scheduling into a variable delay region; the delay region can only be filled up to the minimum number of slots. On a high issue-width processor, such as sixteen instructions, the minimum delay region size is markedly lower than the average delay region size. This means that using local delay region scheduling will cause a high number of wasted clock cycles and actually results in more power consumption than just using the dynamic predictor. Hence, local delay region scheduling must be disabled for the sixteen instruction issue width experimental model.

The branches whose dynamic predictor accesses are reduced by the local delay region scheduling account for around 30% of the dynamic instruction stream in the EEMBC benchmarks. Disabling this part of the combined algorithm immediately reduces its effectiveness by this amount. However, since the branch types that use the local delay region are unconditional absolute branches, there is a method that permits a way around this problem. GCC, and other modern compilers, allow for the compiling of position independent code [74]. Used for library creation, this compilation technique specifically removes unconditional absolute branches by replacing them with one or more PC-offset branches. Offset branches are the type that can be fully utilised by the profling and hinting techniques in the combined algorithm.

The simulation results for the sixteen instruction issue processor were generated by compilation with the GCC position independent code technique. Although this seems to alter the terms on which the simulations are carried out, only a small alteration occurs. The branches normally scheduled by the local delay region will now become available to the profiling and hinting techniques.

7.6.1 Benchmark Breakdown

Table 7.8 shows the resulting change in the described metrics for each benchmark on the sixteen-way issue baseline processor after applying the combined algorithm to the entire EEMBC suite. The graph in Figure 7.5 shows the global power saving metric compared with free/ideal branch prediction.

The results for the sixteen instruction issue processor, in Table 7.8 and Figure 7.5, show a similar pattern, in principle, to the previous two baseline architectures. The most successful power savings are, again, in benchmarks: canrdr01,

Table 7.8: Benchmark breakdown results for sixteen-way issue baseline processor

All results are percentages

Benchmark	Dynamic Branches	Hint Rate	Access Reduction	Stream Change	Cycles Change	BP Power Saving	Global Saving
a2time01	22.14	28.38	10.85	-1.06	-1.20	11.71	1.45
aifftr01	20.19	27.59	39.55	-1.06	-2.59	41.86	3.61
aifirf01	18.62	26.53	17.26	-0.57	0.47	23.41	0.45
aiifft01	20.48	25.91	44.18	-1.68	-1.13	41.60	2.35
basefp01	20.60	22.94	20.65	0.18	0.15	26.95	1.32
bitmnp01	20.93	19.55	31.78	-1.59	-1.08	28.85	1.69
cacheb01	21.43	17.65	35.51	-0.77	-1.81	39.07	2.88
canrdr01	33.29	18.99	40.97	-2.26	-5.04	42.95	7.04
idctrn01	13.37	16.97	35.79	-0.59	-0.39	39.42	2.08
iirflt01	18.86	37.50	12.33	-1.37	-0.76	16.43	1.55
matrix01	20.49	17.82	50.87	0.00	-0.52	53.70	4.45
pntrch01	29.20	26.37	44.49	-1.49	-2.90	48.28	5.52
puwmod01	33.63	13.73	41.44	-2.25	-5.39	43.35	7.35
rspeed01	32.75	20.83	22.05	0.81	-2.10	25.17	2.33
tblook01	29.30	30.11	17.74	0.60	-0.91	21.44	1.27
ttsprk01	33.24	11.48	40.06	-2.12	-4.77	42.09	6.67
cjpeg	13.53	30.97	83.57	-1.14	-1.91	85.80	5.19
djpeg	21.25	32.39	39.25	0.25	-1.98	46.37	3.70
rgbcmy	32.47	24.20	47.08	-3.34	-2.15	44.24	4.48
rgbhpg	20.97	19.26	45.43	-1.04	-3.09	50.12	4.54
rgbyiq	24.99	24.03	44.20	-1.75	-3.50	48.33	5.27
ip_pktcheck	31.93	6.01	26.92	0.68	-2.84	29.51	3.82
ip_reassembly	29.52	12.34	49.07	-3.23	-3.20	44.78	4.62
nat	32.23	12.84	48.93	-4.66	-3.17	45.21	5.29
ospfv2	34.88	6.19	64.26	-2.13	-2.00	64.78	7.67
qos	30.28	12.72	63.98	-1.07	-5.14	66.89	9.46
routelookup	34.63	5.84	29.33	0.65	-2.33	33.18	4.07
tcp	16.64	12.13	10.02	-0.96	-1.29	6.83	1.14
bezier01	18.15	17.24	38.97	-1.24	-2.88	39.27	3.69
dither01	20.09	18.54	29.55	0.05	-0.72	32.38	2.54
rotate01	36.95	44.29	70.81	-4.39	-7.18	67.04	11.58
text01	30.66	33.24	37.99	-2.65	-7.04	33.10	7.73
autcor00	9.43	19.58	78.55	-0.37	-0.33	84.88	2.93
conven00	21.78	20.14	37.68	0.19	-0.98	38.44	2.53
fbital00	16.02	20.00	47.47	-0.15	-0.31	49.79	3.07
fft00	15.73	22.35	27.32	-0.83	-0.40	23.77	1.12
viterb00	10.36	18.24	49.24	-0.13	-0.42	59.18	2.38

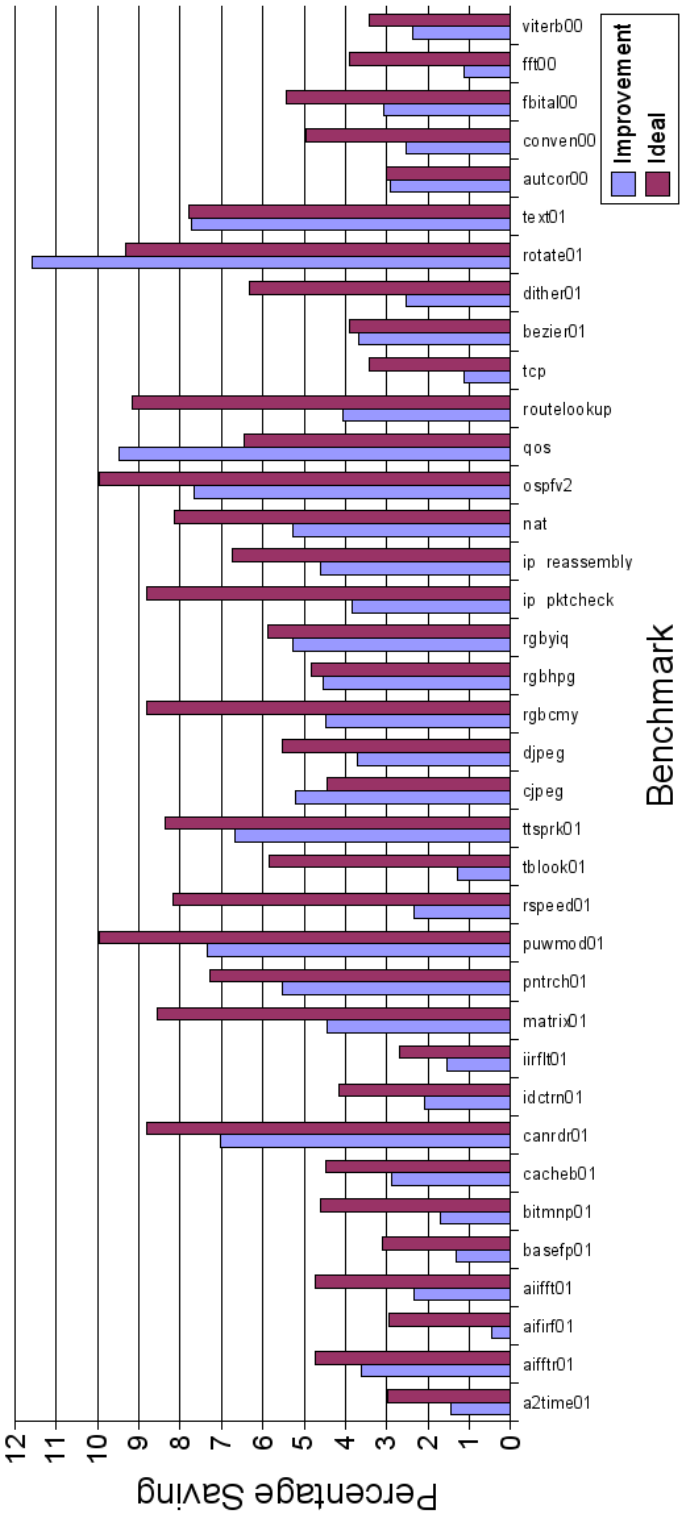

Figure 7.5: 16-way issue baseline global power savings (%) compared with ideal (free) prediction

matrix01, pntrch01, puwmod01, rspeed01, ttsprk01, rgbcmy, ip_pktcheck, ospfv2, routelookup and rotate01. The highest power saving provides an interesting result: rotate01 registers a power saving of 11.58%. This is actually greater than the 'ideal' power saving result (due largely to the decrease in the number of execution cycles). The static hint rate has remained approximately the same as previous baseline models (see the use of position independent code). As a consequence, dynamic branch predictor access reduction is relatively similar. An important point to note is the general trend for benchmarks to register yet more significant instruction/cycle execution reductions.

7.6.2 Averages

Table 7.9 shows the weighted averages of the previous scalar results for each of the EEMBC sub-suites. The graph in Figure 7.6 shows the same scalar averages compared with free/ideal branch prediction.

Table 7.9: Average benchmark results for sixteen-way issue baseline processor
All results are percentages

Benchmark	Dynamic Branches	Hint Rate	Access Reduction	Stream Change	Cycles Change	Power Saving
Automotive	23.64	21.97	41.06	-1.31	-2.39	3.92
Consumer	24.63	30.70	48.03	-1.68	-2.38	4.54
Network	31.27	10.52	49.09	-1.82	-3.87	6.63
Office	25.35	30.76	43.57	-1.61	-3.85	5.59
Telecom	14.24	20.10	45.86	-0.23	-0.45	2.51
Overall	25.68	18.20	47.77	-1.68	-2.74	4.96

The first impression from examining the average results, shown in Table 7.9 and Figure 7.6, is that the sub-suites have come closer again to achieving the ideal power saving metric. The overall stream/cycle change of -1.68%/-2.74% is very significant and can be seen as something of a surprise; it was possible that any negative effects on accuracy might be amplified by such a high issue width processor. Additionally, these results show that the use of position independent code is a suitable alternative, where necessary, to the use of local delay region scheduling.

7.7 Overall Analysis

All three sets of results for the combined algorithm are very encouraging. They show, under the circumstances of the three baseline models, that the combined algorithm will always save power. The amount of power saved can vary greatly between benchmarks, but the overall averages for each sub-suite consistently show significant power savings. The overall success seen in the average results can be

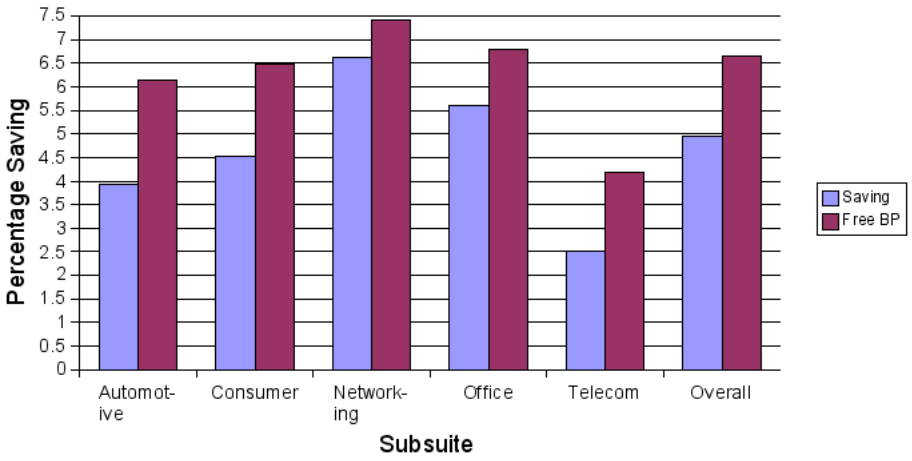

Figure 7.6: 16-way issue baseline average power savings compared with ideal (free) prediction

attributed to the high power savings in the benchmarks that execute for a greater number of instructions (for instance in the networking sub-suite). Indeed, longer executing benchmarks tend to have many tightly iterating loops that are ideal candidates for dynamic prediction removal by the combined algorithm.

One of the first general trends that becomes apparent after examining the results for all three baseline models is that a given dynamic predictor access reduction level does not guarantee the same overall power saving across different benchmarks. This is because, for the branches that are removed/hinted by the combined algorithm, the basic block size varies. The relative power saving will be higher in programs where the basic blocks are smaller; a smaller average basic block size means that branch instructions account for a higher percentage of the dynamic instruction stream. This can be seen best in benchmarks like 'cjpeg' and 'djpeg' which have a very high dynamic access reduction, but a low percentage of the dynamic stream is accounted for by branch instructions. The results is a lower overall power saving compared to 'less successful' benchmarks such as 'ospfv2'.

A surprising, and very important, trend in the results was quite unexpected: most benchmarks register a *decrease* in the number of instructions and cycles executed. As well as counting towards the overall power savings shown in the results, this behaviour is also significant for the performance market. In terms of power saving, the a decrease in the size of the dynamic stream is more significant as it means that fewer instructions actually utilised the control logic inside the processor. However, the change in the number of cycles is also significant as this affects the actual time for which the benchmark executes (even when not executing instructions, a processor will still consume power). The reason behind the reduction in the number of instructions/cycles executed is because the combined algorithm actually increases overall prediction accuracy by providing more accurate predic-

tions for some difficult-to-predict branches.

The scalar baseline model shows an overall power saving of 5.92%. The number of instructions/cycles executed remained relatively unchanged. This is an encouraging result in a processor where the ceiling value of branch prediction power is around 10% of global processor power. The dynamic branch predictor in the scalar model accounts for the highest proportion of the processor's global power when compared to the two-way or sixteen-way issue baseline models. This would seem to indicate that the highest overall power savings were to be made here. However, this is not the case: the two-way issue baseline model registers a slightly higher average global power saving at 6.25%.

Although the dynamic branch predictor accounts for a smaller proportion of global power consumption in the two-way issue processor (due to the rest of processor being 'larger'), and the hint rate has remained relatively unchanged, the increased power savings can be attributed to the combined algorithm's static hinting of dynamically difficult to predict branches. In a multiple instruction issue processor, mispredictions are very costly. The combined algorithm ameliorates this problem, and as such shows an increased reduction in dynamic predictor accesses and the dynamic instruction stream. This results in an increased overall power saving when compared to the scalar baseline model.

Finally, the sixteen-way instruction issue shows the most surprising results set. Initially, it was suspected that the combined algorithm might result in a very slight reduction in overall prediction accuracy during a program's execution. It was expected that on scalar, and small issue, pipelines any slight decrease in accuracy would be offset by power savings of the reduction in dynamic predictor accesses. It was therefore possible that the sixteen-way instruction issue processor would exacerbate any decrease in prediction accuracy by its huge misprediction penalty. In actuality, the results show that the combined algorithm actually improves prediction accuracy and shows almost a 3% reduction in execution time (cycles). This result is significant for the performance market irrespective of any power saving results. The overall power saving is 4.96%; while this saving is lower than on the other baseline models, it is due to the branch predictor accounting for a much lower proportion of overall power consumption as the processor logic is vaster and more complex. Additionally, the dynamic predictor access reduction is much lower on this architecture. This is because the unhinted branches account for more dynamic predictor accesses on the sixteen-way baseline as a consequence of the misprediction window (up to 3 branches may needlessly access the dynamic predictor on a misprediction.

7.7.1 Results Summary

The results show a significant power saving on all architectures. The savings range, on average for each baseline, from between 4.96% to 6.25%. The algorithm was marginally most successful on the two-way issue baseline model. The slightly

surprising behaviour shown in the results is that the combined algorithm has key performance implications for MII, resulting in an execution speed up of almost 3% in the sixteen-way baseline model.

The questions remaining after this analysis are:

- How does the profiling/hinting regime, using adaptive bias measurement, compare to a fixed bias level or compiler hinting heuristics?
- Since certain branch addresses are permanently removed from dynamic predictor access during a program's execution, does this decrease aliasing in the predictor? As a consequence, can set associativity in the power-hungry BTB be reduced?
- Are these results applicable to a 'real' instruction set?

The next chapter is dedicated to exploring the above questions by empirical investigation and comparison to existing technologies.

Chapter 8

Comparisons and Enhancements

There are various comparisons and enhancements that appear logical after the previous experimentation with the combined scheduling and hinting algorithm. This chapter explores the most prominent of these modifications using further experimental results. Although relatively short, this chapter is important as it demonstrates the possible further gains available from 'tweaking' hardware elements further, and how the ABBM system compares with more traditional schemes.

8.1 Comparison of ABBM with Fixed Bias Level and Compiler Heuristics

One of the key novelties of the combined algorithm is the use of the adaptive branch bias measurement, used to decide whether to insert a static behaviour for a branch based on the dynamic predictor's accuracy.

The use of this method, and profiling in general, raises two significant questions that remain hitherto unanswered:

1. How does using an adaptive branch bias measurement, with profiled dynamic branch predictor behaviour, compare to the use of a simple fixed bias level for each branch?

2. How does using an ABBM compare to 'intelligent' compiler heuristics that determine likely biased branches?

The answers to these questions are linked. The answer to question 1 can be achieved by simply 'turning off' adaptive branch bias measurement, and replacing it with a single, fixed bias level (not the ABBM). The results for this will also provide a rough approximation for the answer to question 2. This is because a compiler that uses heuristics to determine biased branches will inevitably approximate

to a fixed bias level; it can only, by virtue of the absence of execution information, determine whether a branch is 'likely very biased'.

The fixed bias level used to generate the results shown in this section is 85%. The baseline model used is the scalar processor. This means that if a dynamic branch tended to one direction for at least 85% of its dynamic occurrences then it would be assigned a static prediction hint in that direction. This bias level was chosen as it was thought to be high enough so as to minimise interference with the accuracy of the dynamic predictor, but not so high that too few branches would be found for hinting. A higher bias level was found to be too fine-grained to apply a significant number of hints.

8.1.1 Results and Analysis

Table 8.1 shows the average sub-suite results generated using the fixed bias level of 85%.

Table 8.1: Average benchmark results for fixed bias hinting
All results are percentages

Benchmark	Dynamics Branches	Hint Rate	Access Reduction	Stream Change	Cycles Change	Power Saving
Automotive	23.63	22.05	46.87	2.98	2.14	0.51
Consumer	24.63	28.17	51.63	2.17	2.81	0.06
Network	31.27	9.99	45.18	-1.16	-2.92	1.12
Office	25.35	28.30	53.94	1.00	0.14	1.51
Telecom	14.14	20.23	57.96	2.16	1.69	1.55
Overall	25.67	17.32	50.30	1.58	2.51	1.04

These results show that using a fixed bias level to assign static hints, which is then used in conjunction with a dynamic predictor, is probably not a sensible idea. It can be seen that although dynamic predictor accesses are reduced, on average by 50%, there is a significant associated penalty. Overall, the average number of instruction executed increases by 1.58%, and the number of cycles executed increases by 2.51%. In terms of performance these figure are extremely significant, and detrimental. The overall effect observed in power is that there is a saving of about 1%, but it is not particularly significant.

The reason for this ill-effect is that the set of static branches hinted using a fixed bias level will be significantly different in content when compared to those chosen by the adaptive method. All branches that lie over the bias level will be assigned a hint. However, many of these branches will quite possibly be almost perfectly predicted by the dynamic predictor, and thus the fixed bias will decrease prediction accuracy for the branch. This argues for a very high fixed bias, but this means that fewer and fewer branches are available for hinting. Additionally, the ABBM has the advantage of hinting difficult-to-predict branches. Although the fixed bias will

hint some branches that are heavily biased and difficult to predict, it will not hint branches that are not heavily biased, but would still benefit from a static prediction.

The effects observed in this experiment also suggest that a similar effect is likely to arise from the use of static analysis of the assembly code by a compiler. This will depend on the heuristic used, but it is very difficult to determine, from static code, which branches would be suitable to issue a static prediction.

These results show an overall power saving of 1%, but it is possible that this is an artefact of the power metrics derived from the Wattch Simulator. It is suspected that Wattch does not assign a sufficiently significant portion of the energy model to static power dissipation during inactivity. This possible artefact does not affect the validity of the main body of results because the the number of cycles actually decreased (in fact, the results could be even better when this artefact is considered). Such a high increase in the number of cycles executed should make a global power saving very difficult, even with a 50% access reduction to the branch predictor. This would almost mean that the results shown in the previous chapter could be even stronger, given the decrease in the number of executed instructions/cycles. It is, however, possible that the number of instructions executed is more significant than the number of cycles (for power) and, as this was only 1.58%, the savings in branch predictor power outweighed this.

The overall impression from these results is that using the novel ABBM is far more beneficial than a system that uses a fixed bias level (or equivalent).

8.2 Reducing Set Associativity in the Branch Target Buffer

The results from the previous chapter show that, in general, almost 20% of static branches are assigned a hint. This means that around one in five branch addresses no longer access the branch target buffer. Additionally, the branches assigned a static hint are generally highly iterative branches that account for a significant portion of accesses to a dynamic predictor, and have a strong affect on the dynamic stream. If these branches are not accessing the BTB then it is possible that the number of collisions between branch addresses will decrease, and thus it may be possible to reduce the set-associativity in the BTB.

The results in this section show the effect of reducing the set-associativity on the 2-way issue baseline model from four to two. The EEMBC suite was scheduled and hinted as normal using the combined algorithm.

8.2.1 Results and Analysis

The resulting change after altering the set associativity in the BTB can be seen in two result metrics: the resulting change in the number of instructions executed (was prediction accuracy affected?) and the resulting power saving. The effect on these two variables can be observed centrally in the form of the change in the number

of executed instructions. The results show the change from standard application of the combined algorithm. This means all results are measured from the baseline that has already had the combined algorithm applied. Figure 8.1 shows the change in the number of executed instructions. Figure 8.2 shows the change in global power saving (a negative power saving is a power consumption increase).

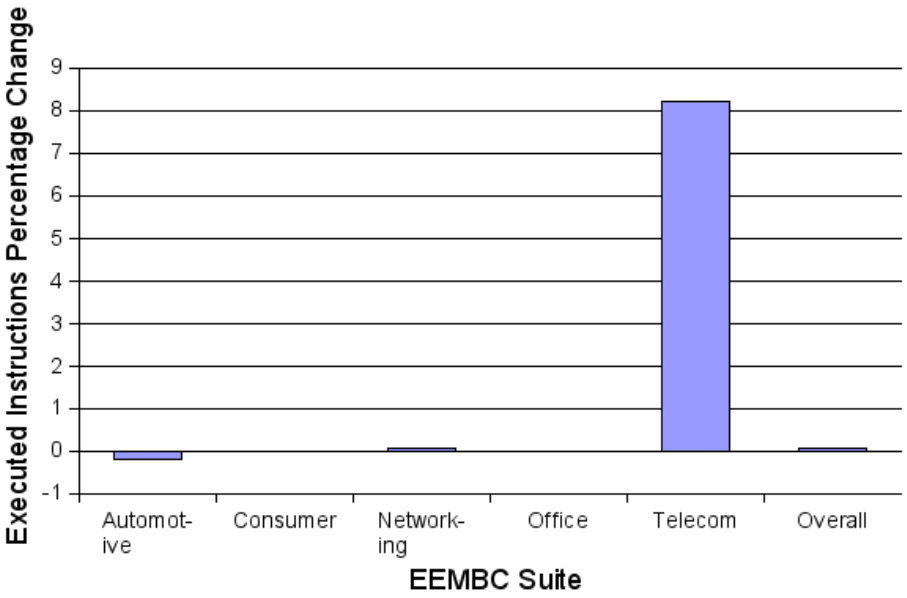

Figure 8.1: Average Change in the dynamic instruction stream after resizing the BTB from four-way to two-way set-associativity

The most important results to examine first are the changes in the number of instructions executed for each sub-suite. It can be seen that, generally speaking, the reduction in size of set-associativity has had little impact on the dynamic prediction accuracy, and thus the number instructions executed. However, there is one exception: the Telecom sub-suite, for which there is a major increase in the number of instructions executed. This is significant, as it shows that it is possible for a negative impact to occur as a result of the reduced accuracy of the target address predictor.

The results for global power saving reflect the changes in the instruction stream. Power savings can be seen with the overall average being slightly under 0.4%. It is significant to note that for the Telecom sub-suite there is an increase in global power consumption as a result of decreasing the associativity in the BTB.

Although these results show that, overall, additional power can be saved by altering the configuration of the BTB, they also demonstrate that doing so removes the ‘always-a-win’ nature of the combined algorithm and mean that it is possible for bigger problems to occur when compared to the benefits. Whether making these alterations to the BTB will result in an additional power saving depends entirely on which static branches are assigned hints.

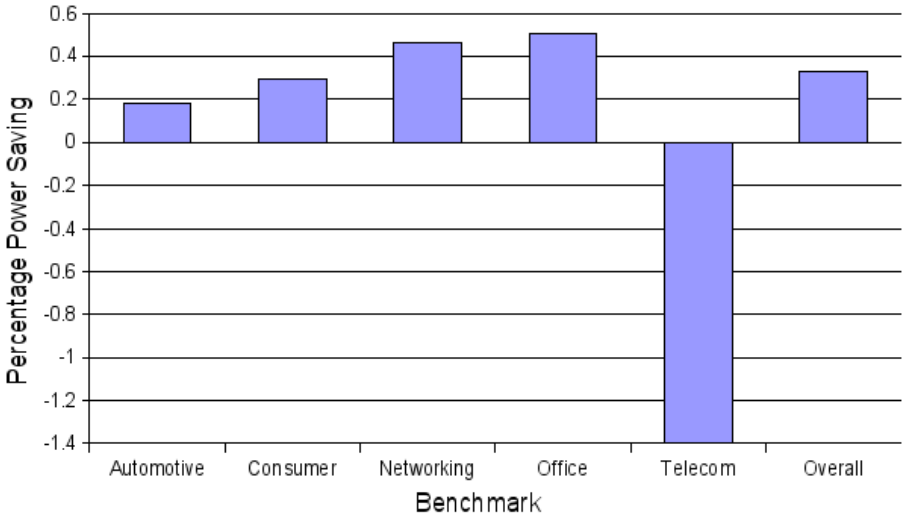

Figure 8.2: Additional power saving after resizing the BTB

It is likely that altering the hardware is not sensible, for two reasons: firstly, it is possible for a negative result to occur when the combined algorithm is used, and secondly, once such alterations to the hardware are made, it demands the use of the combined algorithm for all software. One usual advantage of the combined algorithm is that is it relatively light in hardware implementation and does not necessarily have to be used. Making the hardware dependent of its use removes this advantage.

8.3 Summary

This chapter, although relatively small, has explored two significant questions which arise from the main body of results shown in the previous chapter. The initial question was concerned with whether or not the ABBM was better than a fixed bias hint assignment method. The results show that, in terms of energy, the ABBM gave much more favourable results. In terms of performance, the use of a fixed bias level was clearly detrimental. It is likely that, in some architectures, this performance impact could dramatically increase the global processor power consumption as well.

The second question addressed was whether the BTB could be reduced in size/associativity, since the combined algorithm significantly and permanently cuts the number of branch addresses entering the BTB. The results show that it is possible to reduce from four-way to two-way set associativity without having a significant affect on the number of instructions executed (which is central to not causing power incursions). However, the telecom sub-suite registered a significant increase in the number of instructions executed. This means that hardware designer needs to

decide whether the meagre additional power savings of reducing set associativity are worth the risk of increasing execution time (and also power consumption).

The next, and final, chapter discusses in depth the findings of the experimentation chapters and the investigation in general.

Chapter 9

Conclusion and Discussion

The previous two chapters have presented and discussed the results of using the combined algorithm on a simulated architecture, and several baseline models. These results were also then compared with the success of other methods and their applicability to real architectures. This offers a summary of the key novelties presented in this work, the criticisms that could be levelled against it and where the results lie in the broader research area. Finally, the conclusions drawn from these experiments are stated with the aim of leading into the future work.

9.1 Thesis Summary

This dissertation has proposed, and investigated, an approach to increasing the energy efficiency of a dynamic branch predictor in a modern embedded processor. The approach taken combined several traditional methods: local delay region scheduling, profiling and hint bits, however the combination is novel. The central aim of the combined algorithm is to reduce the dynamic branch predictor activity factor (by reducing the number of accesses to it) without increasing the execution time of the program by negatively affecting the dynamic branch predictor's accuracy. This is achieved by bypassing the dynamic predictor for certain selected branch instructions.

The local delay region can easily be used for unconditional absolute branches, since they will always be taken to a fixed target, and profiling can be used to generate static prediction data for offset branches (provided they share a single branch format). The profiling mechanism uses an adaptive system of measurement to decide whether or not to assign a static prediction to a branch by comparing its dynamic bias to the dynamic branch predictor's accuracy for that branch. For branches more biased than their dynamic prediction, a static prediction is assigned. This ensures minimum impact on prediction accuracy, which is itself crucial for

global processor energy efficiency. In fact, an increase in prediction accuracy was measured.

The combined algorithm is supported by simple hardware modifications in the instruction fetch and execution pipeline stages where the hint bits are used to prevent a predictor access from occurring. In the course of this research, results have shown that around 63% of dynamic branch predictor accesses can be avoided on accurate (large local [PAg]) branch predictors, and this will have negligible impact on execution time. In many instances, execution is even reduced by the static prediction of dynamically difficult-to-predict branches. On a high powered embedded processor, around a 6.2% global power saving could be expected. This saving can be higher when the dynamic predictor is less accurate as more branches will be hinted.

9.1.1 Key Novelties and Contributions

Although some of the methods used in this project already exist, there are several key novelties presented in this thesis:

Combining Techniques – Static and dynamic techniques can be further combined into a comprehensive system. Delay region scheduling has not been used in conjunction with a dynamic predictor in this way before, and this combination is therefore a novel approach.

Adaptive Branch Bias Measurement – Assigning static branch predictions has, until now, been conducted using either compiler heuristics or profiling with a fixed bias level. In both situations, the static prediction assigned is given no guarantee as to how it will affect the accuracy of that branch's prediction. Measuring the branch bias adaptively ensures that a static prediction is only assigned for a branch where it will not adversely affect that branch's prediction accuracy. This technique is a significant and novel contribution.

Power Saving Hint Bit Logic – Although the use of hint-bits is by no means new, using them to control the switching activity of a branch predictor has not been fully researched. When used in the proposed novel way, simple logic switches on/off the datapath of the branch predictor control logic by using the information in two simple hint-bits. For hinted taken branches, the proposition is essentially to include a very simple 'speculative' decode, of a hinted taken branch instruction, in the instruction fetch pipeline stage.

9.2 Generalisation

The previous two chapters presented results for the combined algorithm using various different baseline configurations. However, these results do appear tied to those

architectures. A logical question from any processor designer is "how much power can be saved on my architecture using this algorithm?" This section aims to generalise the power saving potential of the combined algorithm into an approximate equation. It should be noted, however, that the sheer number and complexity the of variables involved makes it difficult to give a precise estimate, and the equation should be considered indicative only.

Equation 9.1 shows the relationship between global power saving and the effectiveness of the algorithm.

$$G \approx R \times P \tag{9.1}$$

Where:

G = Global Power Saving. The power saved across the entire processor by the application of the combined algorithm to the hardware and compilation process.

P = Branch predictor power consumption as a fraction of global power. The proportion of global processor power accounted for by the branch predictor unit. This is affected by both software and hardware. In the hardware, the relative size of the branch predictor compared to the rest of the processor is key. In the software, the average number of branch instructions in the dynamic stream will affect how often the branch predictor is accessed on average. Smaller basic blocks mean more frequent branch predictor accesses.

R = Fractional reduction in dynamic branch predictor accesses as a result of the combined algorithm. The number of dynamic accesses that are avoided is decided by the number of dynamic branches that are assigned a static hint to use either local delay region or an assumed direction. This is, in turn, decided by the number of branches/branch types available for hinting in a given instruction set. The number of branches available is determined by the dynamic total of the number of unconditional absolute branches and the number of offset type branches. The proportion of these branches that are assigned a hint depends upon how biased each branch is compared to the kind of dynamic predictor being used in the existing architecture.

The availability of instructions in the instruction set in the experimental architecture used in this project was around 70% of the dynamic stream. From this, using an accurate local predictor, around 65% of *all* dynamic branch predictor accesses were avoided through scheduling/hinting. Depending on the accuracy of the dynamic branch predictor in use, this figure can vary between avoiding 60% and 70% of all dynamic branch predictor accesses; a less accurate predictor results in a higher proportion of instructions that can be assigned hints as their bias is comparatively greater than the prediction accuracy in more cases.

Hence, if we assume that most instruction sets allow, on average, for 70% of all dynamic branches to be subjected to the use of the combined algorithm, and this

results in, on average, an overall reduction of 65% of dynamic predictor accesses then:

$$G \approx 0.65 \times P \tag{9.2}$$

If, for example, the dynamic branch predictor accounts for around 10% of global power consumption then the combined algorithm could save 6.5% of global processor power dissipation (Equation 9.2). This is both interesting and significant.

The factors in how effective the combined algorithm will be are highly variable and cannot easily be estimated. But as a rough guide, and with the various baseline models tested, this equation gives an engineer a good idea of the potential power savings.

9.3 Critique

The combined algorithm discussed and presented in this dissertation has several aspects, both in hardware and software implementation, that raise the possibility for discussion. This section aims to address these criticisms and counter them with an explanation of why the particular aspect is not necessarily the problem it appears to be, or by demonstrating how the algorithm can be modified to work around the particular problem.

9.3.1 Local Delay Region

Local delay region scheduling can attract a great deal of consternation from veteran compiler designers. This is generally because of the following criticisms:

1. With well optimised assembly code, it is difficult to find instructions in a basic block that do not affect the outcome of the branch direction.

2. To make matters worse, modern processors have many pipeline stages before branch resolution. This means that there are many instruction delay slots to fill.

3. Superscalar processors do not have a fixed delay region due to dynamic instruction scheduling, and as such the number of delay slots is dynamically context sensitive.

The counter argument to these criticisms is the way in which local delay region scheduling is used by the combined algorithm in the implementation specified. In this thesis, the local delay region is used only for certain branches. It is no longer burdened as being the sole method of delay region resolution. Furthermore, in this research, the local delay region is used only for unconditional absolute branches.

This type of branch is not dependent on *any* of the preceding instructions in the basic block. As such, it is possible to find local candidate instructions for scheduling. The number of candidate instructions, in this situation, is limited only by the size of the basic block being scheduled.

Superscalar processors do present a distinct problem for using local delay region scheduling. However, this is not (always) the problem that is seems to be. Most embedded processors are not superscalar, and since the target of this algorithm is the embedded market, this algorithm is well suited. Furthermore, particularly in superscalar processors with a relatively small issue width, the dynamic scheduling algorithm can be configured such that there will always be a minimum number of delay slots (and, in many processors, this is already the case). Most branches have a delay region which, through profiling, can be observed to have a size which almost always conforms to the minimum delay region size. As a result of this, instructions can be scheduled into the delay region up to the minimum number of slots without having a negative affect on performance. In order to maintain program semantics, the delay region, in this situation, must end with a HLT or padded with NOPs in case the delay region in a particular context is larger than the minimum size, and therefore stop the processor executing instructions from a speculated path.

9.3.2 Hint Bits

Introducing new bits into an instruction set/format may cause concern among architecture designers. The combined algorithm requires two additional bits in all branch instructions. Although there is relatively little opposition to using redundant bits in branch instructions, there remains one typical criticism:

- **The use of two hint bits in only branch instructions requires, and assumes, that there exists predecode logic in the instruction fetch stage to discover branch instructions. How can you justify this?**

The response is relatively straightforward: the combined algorithm, through its hint bits, is focused on reducing the power consumption of a dynamic branch predictor. In a power sensitive architecture, with a dynamic branch predictor, the first thing that should be done is to include predecode logic to disable branch predictor accesses for non-branch instructions. Otherwise every fetched instruction performs a branch predictor access; this makes no sense for a power sensitive architecture. In fact, the logic required to identify a branch instruction is not complicated in a well designed instruction set. The cost of introducing the ‘predecode logic’ is highly likely to be offset by the associated power savings.

9.3.3 Timing Issues

To make full use of the two hint-bits included in branch instructions, the branch predictor access must be performed in series with the i-cache access. This is because the hint bits are used to control whether or not to access the branch predictor, and these are not known until the i-cache access is complete. Some modern high performance architectures access the branch predictor in parallel with the i-cache to reduce the clock window in the instruction fetch pipeline stage. This can be accomplished as no contents of the instruction being fetched are required to access the branch predictor (only the program counter value). These two timing scenarios are shown in Figure 9.1 below.

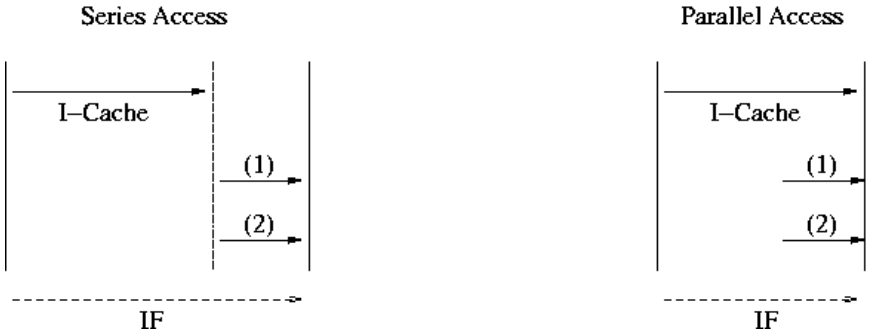

Figure 9.1: Series and parallel i-cache/branch predictor access. (1) and (2) represent the direction and target address predictors, respectively

The series access shown in Figure 9.1 is required by the combined algorithm. However, this problem is not necessarily significant. A parallel access is only necessary on processors that derive their performance from very high clock rates. Fortunately, most power-aware systems to do not function at extremely high clock rates. This means that a series access is perfectly acceptable for most power aware scenarios.

If a parallel branch predictor is absolutely necessary then there is still a relatively simple modification that can be made to the hardware implementation. The modification is the inclusion of a very small direct mapped cache, perhaps as few as eight entries, that are indexed by the program counter and furnish hint-bits from recently accessed branch instructions. This small cache would need to be updated at some stage with a branches hint bits. This cache can then be accessed quickly and in parallel with the i-cache to decide whether to access the full branch predictor. This would, of course, reduce the effectiveness of the combined algorithm, but allows it to be introduced into timing-sensitive processors. The use of such a cache in this way could be seen as a statically initialised version of the Prediction Probe Detector. The use of the combined algorithm in conjunction with the Prediction Probe Detector is discussed in further detail in the Future Work section.

9.3.4 Profiling Duration

A particular criticism that is often levelled at algorithms which make use of profiling is that profiling is generally a very slow process and consumes a large amount of time and resources. In the context of the combined algorithm, the question is: "how can this be justified?" In fact, the question is usually asked when the purpose of the combined algorithm is not completely understood.

The combined algorithm only uses profiling at one point in the production stage of a piece of software. The central aim of the combined algorithm is that it utilises profiling data only once, and then the resulting hinted executable is dispatched to multiple end users and the profiling is not carried out again. Profiling does not need to be carried out again because the combined algorithm ensures that only branches that are better served by a static prediction are assigned one. Because of the broad profiling data sets this ensures that the true data dependent branches are left for dynamic branch prediction.

A further argument against the duration of profiling is that the target audience of the combined algorithm is the embedded market. In this market the execution time of programs tends to be much lower and profiling is far less time consuming. Additionally, it is not stipulated that the combined algorithm must be applied to a piece of code, but rather the processor supports it if the software producer decides (sensibly) to make use of it. Unprofiled/unscheduled code will simply be executed in the normal, dynamically predicted way.

9.3.5 Profiling on a 'Real' Architecture

The profiling results for the implementation of the combined algorithm presented in this dissertation were produced using a simulated architecture in a software processor simulator (HWattch). This has attracted some criticism as the simulation of an entire processor is often considered slow. In fact, simulation is actually a very fast way to retrieve profiling information, particularly the kind of profiling information required by the combined algorithm.

The combined algorithm needs only to record the following information during a program's execution:

1. The direction taken by each static branch at each dynamic occurrence
2. The dynamic predictor's predicted direction for each dynamic branch occurrence

The most important aspect of these results is that it must be possible to directly map the dynamic branch execution history to the static assembly code of the program. This is necessary in order to realise which dynamic behaviour corresponds to which branch, and hence which assembly instruction should be assigned a static hint. Additionally, it must be remembered that, in contrast to most profiling schemes, the behaviour of the dynamic branch predictor is required. This behaviour cannot be inferred from the actual semantic behaviour of the program.

Full Simulation

Achieving the required profiling results in a full processor simulator is relatively straightforward. This is because, by definition, all elements of a simulator are available for inspection and monitoring. This makes obtaining the required branch prediction information simple.

Mapping dynamic behaviours to static instructions was achieved in simulation by annotating each static assembly instruction with a unique identifier which was recorded with each dynamic branch direction result. This simplified the process of profile mapping and enabled reliable branch instruction annotating.

The advantage of full simulation is the simplicity of the solution. The disadvantage is that simulation makes it difficult to simulate a full system, for example a full operating system, and any programs which make use of a fully 'live' system.

Hardware Trace

When profiling results are required under a realistic load, through direct interaction with the operating system or when the program makes use of real hardware, the only viable solution is to profile the program on a 'real' system/processor.

This can be achieved with the use of a runtime tracer, such as HTracer, which can fully log the behaviour of an entire program by executing one instruction at a time and stall the traced program after each instruction has been executed. This process is straightforward and well documented. The difficulty is mapping the dynamic program trace back to the original assembly code, since the program counter value does not necessarily indicate which instruction from a static assembly source file is being executed. To overcome this problem, an additional instruction should be inserted in the assembly code before each static branch instruction that needs to be traced dynamically. This instruction would store an absolute value in a register that indicates which part of the static source code is being executed. This is then interpreted post-trace and used to map profiling data to the static code.

The key difficulty on a hard architecture is that the combined algorithm requires information about the behaviour of the dynamic prediction logic. This is unlikely to be available through the architecture dynamically. The simplest way to generate this part of the profiling data is to take the previous compacted branch trace of the program and parse it through a simple simulator that only models the branch predictor used in the hard architecture. This can then produce the desired branch predictor information for each dynamic branch in the unrolled program execution contained in the compacted trace file.

Profiling on a real/hard architecture allows full use of the system in the profiled program and a full operating system. It can be used to obtain more useful results than perhaps are available in fully simulated environment.

9.3.6 Dependency on Datasets

The final significant point of contention over a system such as the combined algorithm is its use of 'strong' static hinting. That is to say, the combined algorithm uses hint bits, for certain branches, to reflect a static prediction that overrides all dynamic prediction for that branch. The argument is that such a system could significantly reduce the overall accuracy of branch prediction when the hinted program is executed with a different dataset. Such an argument has a valid basis, but can be seen to be unfounded in the case of the combined algorithm.

The first major point which must be considered is that all systems of profiling, in order to be valid, must be carried out with an adequate dataset. This could either be a significantly large dataset or several different datasets. The combined algorithm uses this data to assign hints for only those branches where the static bias of a branch, across this broad profiling dataset, is greater than or equal to the dynamic predictor's accuracy for that branch. This process essentially assigns static predictions for only heavily biased branches and difficult-to-predict-dynamically branches. These branches are specifically less dependent on datasets and thus assigning them a static prediction will not, and does not, affect overall prediction accuracy. This is confirmed by the results presented in this dissertation. All benchmarks were profiled against two different datasets and then tested on a third dataset to generate the actual results. This removed the possibility of over-fitting to a particular dataset. However, through experimentation, the same results can almost always be obtained for this algorithm using a single dataset.

9.4 Related Work Comparison

An important point of discussion, now that the results are known for the combined algorithm, is the significance of this work compared to other related approaches to energy efficient branch prediction.

The combined algorithm essentially treats the dynamic branch prediction logic as a blackbox; it does not interact with the internals of the dynamic branch predictor, but rather interacts with its external behaviour. This is significant when considering related approaches. There are many approaches to energy efficient branch prediction that have focused on designing inherently efficient dynamic branch predictor units. However, these are not actually in competition with the combined algorithm directly as the combined algorithm can be used with almost any scheme of dynamic branch prediction. The only significant piece of related work that it is sensible to compare and contrast with is the Prediction Probe Detector which also attempts to reduce the number of dynamic branch predictor accesses.

9.4.1 Prediction Probe Detector

The PPD, as discussed in Chapter 3, introduces a small cache-like structure into the instruction fetch stage of a processor. This structure was initially intended simply to stop branch predictor accesses occurring for every fetched instruction in an environment where it was not possible to wait for an instruction to be fetched from the i-cache and then use predecode bits to prevent a branch predictor access. The PPD essentially behaved as a control flow instruction predictor; it would predict whether the next instruction to be fetched from the i-cache was a branch instruction, and hence whether the branch predictor should be accessed.

The PPD was then enhanced to predict whether the next instruction was a conditional branch or an unconditional branch. This allowed the control logic to decide whether a direction prediction or just a target prediction was required. This information was stored in two bits (one controlling the direction predictor, and the other controlling the target predictor) in a small cache where each line corresponds to a line in the i-cache. The PPD is then updated once the actual instruction has been decoded and its type is known. The PPD means that the branch predictor does not need to be accessed in every cycle, even in systems where the branch predictor logic has to be accessed in parallel with the i-cache.

Finally, the PPD was further extended to use compiler hints which indicate whether a branch is biased to the extent that it is unchanging. This required an additional bit in the PPD to indicate this new possibility. These hints were relatively crude and only convey information for a small subset of branches that are completely unchanging.

Overall, in the experimental baseline architecture used in the PPD investigation, a global processor powersaving of around 3% was observed.

It was noted as a final idea of the PPD journal paper [82] (published *after* the start of this work) that much bigger power savings would be possible if some form of hints could be used to convey information about branch behaviour to the PPD. This gives reassurance to the methods used in this project, but is still very different from the combined algorithm.

In contrast to the PPD, the combined algorithm assumes a series access with the i-cache/branch predictor (possible in processors with a lower clock speed) and the inclusion of predecode logic. These are not big assumptions, as discussed previously, but do change the premise of the scheme used. In fact, if a parallel access is required of the branch predictor and i-cache, then an approach similar to that described in the future work (9.5.3) of combining the PPD with the combined algorithm would be the most sensible.

All of this means that, in general terms, the power savings achieved by the combined algorithm are in *addition* to those achievable by the PPD. That is to say that most of the power savings achieved by the PPD are already assumed to have occurred in the results for the combined algorithm. The combined algorithm results assume that only branches access the branch predictor logic (the chief power saving achieved by the PPD). This means that, according to the results presented

in this dissertation, significant additional power savings could be achieved by the extension of the PPD with the combined algorithm (where a parallel access is necessary – otherwise the PPD is of no benefit to the combined algorithm).

The way the combined algorithm achieves the additional power savings over the PPD is by the following methods:

1. The inclusion of full static predictions for branches, including the target address for the most common branch type format. This enables not only to completely avoid furnishing predictions for many branches, but also negates the need to update the predictor for those branches (apart from the history register in two-level predictors such as global predictors).

2. Adaptive Branch Bias Measurement through profiling maximises the number of branches that can be assigned a static hint by using a per-branch bias assignment system. This assigns a static prediction for all branches that are more biased to one direction than the accuracy of their dynamic prediction. This has the effect of removing both 'very biased' branches (though not just traditional ones) and difficult-to-predict branches.

3. Local delay region scheduling is used for unconditional absolute branches. This removes accesses for a large subset of dynamic branches.

4. Simple hardware support in the instruction fetch stage allows for very simple calculation of the target address for the most common branch type format. This is significant as the branch target predictor (BTB) is a major consumer of power.

These key differences demonstrate that their are significant advantages that can be achieved over the existing PPD scheme. These advantages could work together with the PPD in a dynamic context.

9.5 Future Work

During the experimentation and design work conducted throughout this project there have been several ideas for future work that have arisen. Some are very closely related to the research conducted surrounding the combined algorithm, but others are broader and more distant in context. This section takes a very brief look at several possible avenues for future research.

9.5.1 Maximising the Fetch Window of Wide Issue Processors

Superscalar processors fetch multiple instructions in each cycle in order to fill a multiple instruction fetch queue. In wider instruction issue processors, such as sixteen instruction issue, there is likely to be more than one branch instruction

fetched in each cycle (assuming that the fetch queue can be somewhat emptied each cycle – the number of function units is also high). This is a problem because each time a branch instruction is encountered, a prediction must be formed by the dynamic predictor as to which direction the branch will take, and, if the prediction is taken, what the target address will be. Branch prediction logic is complex and will generally use of the rest of the cycle time for a predicted taken branch. This means that, potentially, it can be difficult to fill a wide instruction fetch window when more than one branch is encountered per cycle.

The proposal for future work consists of measuring the effectiveness of the hinting scheme described in this dissertation at ameliorating the branch prediction bottleneck for filling a wide instruction fetch window. A dynamic branch prediction is unnecessary for all branches hinted by the combined algorithm. The combined algorithm can hint between 60%-75% of all dynamic branches. This potentially means that a significant reduction in the branch bottleneck could occur and that the instruction fetch window could be significantly more successfully utilised.

9.5.2 Hinting Libraries

Due to technical complexities, and design decisions, the libraries used by all of the programs in the experimentation chapters of this report were unhinted. This means that all branches in libraries used dynamic prediction, whether or not they were biased. This means that there is a further source of potentially avoidable dynamic branch predictor access, and therefore a further source of power saving. The difficulty is that libraries, or more specifically shared libraries, are shared in memory between one or more programs. Assigning a static prediction to branches in a shared library, using profiling information, is very difficult. The branches in the library can appear in any number of contexts within several different programs where the static predictions may be highly inaccurate.

A possible solution to this is to statically link a program to the libraries that it uses (or a subset of the libraries that it uses). This would remove the context problem of a library appearing in various different programs. As such, it is then possible to hint the library functions within the target program as normal.

However, statically linking a program to its libraries will increase the size of the program (as the library calls must now be contained within it) and the library will no longer be shared. An investigation would decide whether the overall increase in the size of programs was worth the extra potential power saving that could be obtained from hinting the newly hintable branches.

9.5.3 Combining with the Prediction Probe Detector

The combined algorithm cannot be used in a processor where the clock rate is so high that the branch predictor must be accessed in parallel with the i-cache using

only the program counter. This is because the combined algorithm relies on the availability of the hint-bits contained within a fetched instruction. These hint bits are then used to decide whether the dynamic branch predictor should be accessed.

An initial solution to this seems to be the use of some kind of very small direct-mapped, or even fully-associative, cache. This would be updated with the hint bits for branch instructions and could double as a method to avoid predicting for non-branch instructions.

The combination of the 'combined' algorithm with the PPD would provide results that would not only be interesting from the point of view of the combined algorithm in parallel access contexts, but also because it would extend the prediction PPD with some of the hinting methods that were suggested as enhancements to the PPD. In fact, the hinting methods used by the combined algorithm are far better than those suggested for the PPD. Such a combination would provide an analysis of an energy efficient branch prediction scheme suitable for high performance processors.

9.5.4 Hints and Context Switching

One area of branch prediction that has not been well examined is the effect of context switching, in a real operating system environment, upon branch prediction. When an operating system performs a context switch, one program is 'switched out' of running execution for another. This means that all of the information in the branch predictor is now irrelevant: any correct predictions from the dynamic branch predictor are now by chance and the branch predictor must begin learning the branch behaviours again. It is possible that context switching has a highly significant effect upon dynamic branch prediction accuracy. The reason that this has not been well quantified is because almost all branch prediction experimentation and design is conducted using simulators. Generally, these simulators only execute one program and do not run an operating system; running an operating system in a simulated environment is difficult and very slow. Therefore, context switching is rarely tested fully.

An interesting proposal for experimentation in the future would be to fully quantify the relationship between context switching (under different loads) on the accuracy of dynamic branch predictors. After this, experimental results of the use of the combined algorithm in a context switching environment should be compared. It seems both logical and probable that, under heavy context switching, the overall accuracy of branch predictions would be higher with the use of the static hints from the combined algorithm as they do not have to be re-learned each time a context switch occurs and they will suffer no resulting interference.

9.5.5 Profiling and Processor-Wide Power Saving

The thesis presented in this dissertation is largely focused on the power savings that can be achieved by the profiling of control flow instructions and the resulting reduction in accesses to the dynamic branch predictor logic. However, this gives motivation for the possible use of profiling and hinting for saving power in other areas of processor design.

Two important factors in the power consumption of modern processors are the clock frequency and static power dissipation. The clock frequency has a linear affect on power consumption by the propagation of the clock signal throughout the control logic. This power drain is ameliorated by control logic that performs clock gating – the disconnecting of the clock signal from idle units. Deciding which units are idle is performed by the clock gating control logic. Static power dissipation can be reduced by the same gating logic, but to gate the entire power supply of units that do not need to constantly maintain a state.

The key point about gating, of both varieties, is that the processor must decide dynamically which units or clocks should be gated. Although the introduction of this technology has saved significant power, it not optimal. Clock gating rarely performs at the theoretically best performance.

An interesting area of investigation would be to use profiling data to insert either hint-bits or hint-instructions which can be interpreted by the processor at runtime in order to gate units or clocks more efficiently. For instance, with only simple compile time analysis, the compiler can tell whether or not certain logic and arithmetic units will be used. This could be represented in a hint instruction which tells the processor to power down the appropriate units, and when to power them back up again. Profling data could then also offer dynamic insight into which areas of the processor are likely to be used at a given period of execution. A hint instruction could be a new type of instruction introduced into the instruction set where one part of its field represents several different elements of the processor, and whether they should be powered up or down (1 or 0). Of course, the processor could work without any such hint information, but the presence of such hint information could be used to enhance/override the dynamic behaviour.

The result of these experiments could be a more efficient and less hardware intensive approach to general processor power savings.

9.6 Concluding Remarks

In conclusion it is appropriate to say that previous related work has not fully realised the possible power savings and energy efficiency that can be achieved when static scheduling/hinting and dynamic branch prediction work in synergy with one another using minor hardware support. When profiling is performed on a target program, with ABBM, a significant number of dynamic branch predictor accesses can be avoided, and the consequent power consumed to perform those accesses can

be saved. In addition to the power savings that are possible, it can also be seen that the system of adaptive branch bias measurement allows overall prediction accuracy to be improved. Adaptive bias measurement, through profiling, has implications to the performance market as well, particularly in architectures which already support directional hinting (for example the PowerPC architecture).

The central message of this thesis, that should remain in the mind of the reader, is that traditional techniques, namely delay region scheduling and profiling, can be used to achieve significant power savings in modern dynamic branch predictors. The addition of the novel ABBM, simple hardware and hint-bit configuration is crucial to the significant effect that the combined algorithm can have for high performance embedded processors.

Bibliography

[1] R. Adams, S. Gray and G. Steven. HARP: A Statically Scheduled Multiple-Instruction-Issue Architecture and its Compiler. In *2nd Euromicro Workshop on Parallel and Distributed Processing*, p. 8. January 1994.

[2] R. Adams and G. Steven. A Parallel Pipelined Processor with Conditional Instruction Execution. Tech. Rep. 107, University of Hertfordshire, October 1990.

[3] R. G. Adams, G. B. Steven, S. M. Gray and G. J. Green. Code Compaction Algorithms for a Multiple Instruction Issue Processor. Tech. Rep. 127, University of Hertfordshire, January 1992.

[4] G. Albera and R. Bahar. Power and Performance Tradeoffs Using Various Cache Configurations. In *Power Driven Micro-Architecture Workshop*, pp. 64–69. ACM, New York, NY, USA, June 1998.

[5] AMD. *Athlon 64 X2 Processor Manual*, 2006.
URL `http://www.amd.com [navigate to manuals]`

[6] S. Amos and M. James. *Principles of Transistor Circuits*. Butterworth-Heinemann, London, UK, 1999. ISBN 0-7506-4427-3.

[7] ARM. ARM11 Reference Manual, 2005.
URL `http://www.arm.com/manual/arm11 [May 2006]`

[8] R. Arns. The other transistor: early history of the metal-oxide-semiconducor field-effect transistor. *Engineering Science and Education Journal*, pp. 223–240, 1998. ISSN 0963-7346.

[9] S. Asai and Y. Wada. Technology Challenges for Integration Near and Below 0.1um. *IEEE*, vol. 85(4):pp. 505–520, April 1997.

[10] R. Bahar and S. Manne. Power and Energy Reduction via Pipeline Balancing. In *28th Annual International Symposium on Computer Architecture*, pp. 218–229. IEEE, Washington D.C., USA, June 2001.

[11] S. Breach, T. Vijaykumar and G. Sohi. The Anatomy of the Register File in a Multiscalar Processor. In *27th Annual International Symposium on Microarchitecture*, pp. 181–190. ACM, New York, NY, USA, December 1994.

[12] D. Brooks. Power Aware Computing Notes. Tech. rep., Harvard University, USA, September 2004.
URL `http://www.eecs.harvard.edu/ dbrooks`

[13] D. Brooks, P. Bose, S. Schuster, H. Jacobson and P. Kudya. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. IEEE Micro, November 2000.

[14] D. Brooks, V. Tiwari and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th annual international symposium on Computer architecture*, pp. 83–94. IEEE, Washington D.C., USA, 2000. ISBN 0163-5964.

[15] J. Bunda, W. Athas and D. Fussel. Evaluating Power Implications of CMOS Microprocessor Design Decisions. In *International Workshop on Low Power Design*, pp. 147–152. ACM, New York, NY, USA, April 1994.

[16] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Tech. rep., University of Wisconsin-Madison, 1997.

[17] D. Burger and J. Goodman. Billion-Transistor Architectures. *IEEE Computer*, pp. 46–49, September 1997.

[18] M. Butler and Y. Patt. An Investigation of the Performance of Various Dynamic Scheduling Techniques. In *25th Annual International Symposium on Computer Architecture*, pp. 1–9. IEEE, Washington D.C., USA, December 1992.

[19] B. Calder and D. Grunwald. Fast and Accurate Instruction Fetch and Branch Prediction. In *21st Annual Internation Symposium on Computer Architecture*, pp. 2–11. IEEE, Washington D.C., USA, May 1994.

[20] B. Calder, D. Grunwald, D. Lindsay, J. Martin, M. Mozer and B. Zorn. Corpus-Based Static Branch Prediction. In *SigPlan*, pp. 79–92. ACM, New York, NY, USA, 1995.

[21] B. Calder, G. Reinman and D. Tullsen. Selective Value Prediction. In *26th Annual International Symposium on Computer Architecture*, pp. 64–74. ACM, New York, NY, USA, 1999.

[22] A. Chandrakasan and R. Brodersen. *Low Power Digital CMOS Design*. Springer, Berlin, Germany, 1995. ISBN 079239576X.

[23] J. Chang and M.Pedram. Register Allocation and Binding for Low Power. In *IEEE/ACM Design Automation Conference*, pp. 29–35. ACM, New York, NY, USA, 1995.

[24] P. Chang, E. Hao and Y. Patt. Alternative Implementations of Hybrid Branch Predictors. In *28th Annual International Symposium on Microarchitecture*, pp. 252–257. IEEE, Washington D.C., USA, December 1995.

[25] P. Chang, E. Hao, T. Yeh and Y. Patt. Branch Classification: A New Mechanism for Improving Branch Predictor Performance. *International Journal of Parallel Programming*, vol. 24(2):pp. 133–158, 1996.

[26] P. Y. Chang, M. Evers and Y. Patt. Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. In *International Conference on Parallel Architectures and Compilation Techniques*, pp. 48–57. IEEE, Washington D.C., USA, October 1996.

[27] Z. Chen, J. Shott, J. Burr and J. Plummer. CMOS Technology Scaling for Low Voltage Power Architectures. In *IEEE Symposium on Low Power Electronics*, pp. 56–57. IEEE, Cambridge, MA, USA, October 1994.

[28] R. Collins and G. Steven. An Explicitly Declared Branch Delay Mechanism for a Superscalar Architecture. *Microprocessing and Microprogramming*, vol. 40:pp. 677–680, 1994.

[29] T. Conte, K. Menezes, P. Mills and B. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *22nd Annual Internation Symposium on Computer Architecture*, pp. 333–344. ACM, New York, NY, USA, 1995.

[30] D. Dobberppuhl. The Design of a High Performance Low Power Microprocessor. In *International Symposium on Low Power Electronics and Design*, pp. 11–16. IEEE, Washington D.C., USA, August 1996.

[31] C. Egan. *Dynamic Branch Prediction In High Performance Super Scalar Processors*. Ph.D. thesis, University of Hertfordshire, August 2000.

[32] C. Egan, M. Hicks, B. Christianson and P. Quick. Enhancing the I-Cache to Reduce the Power Consumption of Dynamic Branch Predictors. p. 31. IEEE Digital System Design, July 2005. ISBN 3-902457-09-0.

[33] C. Egan, F. Steven and G. Steven. Delayed Branches versus Dynamic Branch Prediction in High-Performance Superscalar Architecture. In *Euromicro*, p. 7. IEEE, September 1997.

[34] R. Evans. *Energy Consumption Modeling and Optimization for SRAM's*. Ph.D. thesis, North Carolina State University, July 1993.

[35] J. A. Fisher and S. Freudenberger. Predicting Conditional Branch Directions from Previous Runs of a Program. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Boston*, pp. 85–95. October 1992.

[36] S. Ghiasi, J. Casmira and D. Grunwald. Using IPC Variation in Workload with Externally Specified Rates to Reduce Power Consumption. In *Workshop on Complexity Effective Design*, pp. 1–10. University of Colorado, June 2000.

[37] K. Ghose and M. Kamble. Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation. In *International Symposium on Low Power Electronics and Design*, pp. 70–75. ACM, New York, NY, USA, August 1999.

[38] R. Gonzalez and M. Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEE Journal on Solid State Circuits*, vol. 31(9), September 1996.

[39] D. Grunwald, A. Klauser, S. Manne and A. Pleszkun. Confidence Estimation for Speculation Control. In *25th Annual International Symposium on Computer Architecture*, pp. 122–131. June 1998.

[40] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 3 edn., 2004. ISBN 978-1558605961.

[41] M. Hicks, C. Egan, B. Christianson and P. Quick. HTracer: A Dynamic Instruction Stream Research Tool. p. 10. IEEE Digital System Design, July 2005. ISBN 3-902457-09-0.

[42] M. Hicks, C. Egan, B. Christianson and P. Quick. Reducing the Branch Power Cost in Embedded Processors Through Static Scheduling, Profiling and SuperBlock Formation. In *Advances In Computer Systems Architecture*, pp. 366–372. Springer, Berlin, Germany, September 2006. ISBN 3-540-40056-7.

[43] M. Hicks, C. Egan, B. Christianson and P. Quick. The Static Removability of Dynamic Branch Predictor Accesses in Embedded Programs Through the Use of Adaptive Bias Measurement and Local Delay Region Scheduling. Tech. rep., University of Hertfordshire, June 2007.

[44] M. Hicks, C. Egan, B. Christianson and P. Quick. Towards an Energy Efficient Branch Prediction Scheme Using Profiling, Adaptive Bias Measurement and Delay Region Scheduling. In *Design and Technology of Integrated Systems*. IEEE, September 2007.

[45] M. Hicks, C. Egan, B. Christianson, P. Quick and B. Dickerson. HTracer: A User Guide. Tech. rep., University of Hertfordshire, July 2005.

[46] M. Hicks, C. Egan, P. Quick and B. Christianson. An Introduction to Power Consumption Issues In Processor Design. Tech. rep., University of Hertfordshire, July 2005.

[47] R. B. Hilgendorf, G. J. Heim and W. Rosenstiel. Evaluation of branch-prediction methods on traces from commercial applications. *IBM Journal of Research and Development*, vol. 43(4), 1999.

[48] M. Horowitz, T. Indermaur and R. Gonzalez. Low Power Digital Design. In *IEEE Symposium on Low Power Electronics*, pp. 8–11. October 1994.

[49] P. Horowitz, Paul and Hill. *The Art of Electronics*. Cambridge University Press, 1989. ISBN ISBN 0-521-37095-7.

[50] Z. Hu, P. Juang, K. Skadron, D. Clark and M. Martonosi. Applying Decay Strategies to Branch Predictors for Leakage. In *International Conference of Computer Design*, pp. 442–445. IEEE, Washington D.C., USA, September 2002.

[51] IBM. *PowerPC Instruction Set Manual*. IBM, Austin, Texas, 2005. URL `http://www.ibm.com (no persistent link)`

[52] Intel. *Intel Architecture Software Developer's Manual*, September 2004. URL `http://www.intel.com/`

[53] Intel. *Intel Itanium Processor Manuals*, 2007. URL `http://www.intel.com/manuals`

[54] C. Isci and M. Martonosi. Run-time Power Monitoring and Estimation in High-Performance Processors: Methodology and Experiences. In *Micro-36*, pp. 93–104. IEEE, Washington D.C., USA, December 2003.

[55] K. Itoh, K. Sasaki and Y. Nakagome. Trends in Low-Power RAM Circuit Technologies. In *IEEE Symposium on Low Power Electronics*, vol. 83, pp. 84–87. IEEE, Washington D.C., USA, April 1995.

[56] E. Jacobsen, E. Rotenberg and J. Smith. Assigning Confidence to Conditional Branch Predictions. IEEE 29th International Symposium on Microarchitecture, 1996.

[57] C. Jesshope. Microthreading a model for distributed instruction-level concurrency. *Parallel Processing Letters*, vol. 16(2):pp. 209–228, 2006.

[58] D. Jimenez, S. Keckler and C. Lin. The Impact of Delay on the Design of Branch Predictors. In *33rd Annual IEEE/ACM International Symposium on*

Microarchitecture, pp. 67–77. ACM, New York, NY, USA, December 2000. ISBN 1-58113-196-8.

[59] J. Karlin, D. Stefanovic and S. Forrest. The Triton Branch Predictor. Tech. rep., University of Texas, Austin, Texas, October 2004.

[60] A. Klauser. *Reducing Branch Misprediction Penalty through Multipath Execution*. Ph.D. thesis, Dept. Computer Science, University of Colorado, 1999.

[61] U. Ko, P. Balsara and A. Nanda. Energy Optimization of Multi-Level Processor Cache Architectures. In *International Symposium on Low Power Design*, pp. 45–49. ACM, New York, NY, USA, 1995.

[62] C. Lee, J. K. Lee, T. Hwang and S. Tsai. Compiler optimization on VLIW instruction scheduling for low power. *ACM Transactions on Design Automation of Electronic Systems*, vol. 8(2):pp. 252–268, 2003.

[63] J. Lee and J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, pp. 6–22, January 1984.

[64] M. Levy. The Embedded Microprocessor Benchmark Consortium. Online, 2005.
URL `http://www.eembc.org`

[65] J. Lorch and A. Smith. Software Strategies for Portable Computer Energy Management. *IEEE Personal Communications*, pp. 60–73, June 1998.

[66] S. Malke, R. Hanke, R. Bringmann, J. Gyllenhaal, D. Gallagher and W. Hwu. Characterizing the Impact of Predicated Execution on Branch Prediction. In *27th International Symposium on Microarchitecture*, pp. 217–227. ACM, New York, NY, USA, December 1994.

[67] S. Manne, A. Klauser and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *International Symposium on Computer Architecture*, pp. 132–141. ACM, New York, NY, USA, June 1998.

[68] A. J. Martin, M. Nystrom and P. L. Penzes. ET^2: A Metric for Time and Energy Efficiency of Computation. In *Power aware computing*, pp. 293–315. Kluwer Academic Publishers, Norwell, MA, USA, 2002. ISBN 0-306-46786-0.

[69] H. Mehta. Techniques for Low Energy Software. In *Proceedings of the 1997 international symposium on Low power electronics and design*, pp. 72–75. ACM, New York, NY, USA, August 1997. ISBN 0-89791-903-3.

[70] M. Monchiero, G. Palermo, M. Sami, C. Silvano, V. Zaccaria and R. Zafalon. Low-Power Branch Prediction Techniques for VLIW Architectures: a Compiler-Hints Based Approach. *VLSI Integration*, vol. 38(3):pp. 515–524, 2005. ISSN 0167-9260.

[71] G. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, vol. 38(8), April 1965.

[72] F. Najm. A Survey of Power Estimation Techniques in VLSI Circuits. In *IEEE Transactions in VLSI Systems*, pp. 446–455. December 1994.

[73] S. Narendra and A. Chandrakasan. *Taxonomy of Leakage: Sources, Impact and Solutions*, pp. 1–19. Leakage in Nanometer CMOS Technologies. Springer-Verlag, 2006.

[74] G. Organisation. GCC Position Independent Code. Electronic.
URL `http://gcc.gnu.org/onlinedocs/gcc-3.4.1/gccint/PIC.html`

[75] P. Otellini. 2005 Intel Developers Conference. Presentation – Available Online, 2003.
URL `http://www.intel.com [search for conference title]`

[76] G. Palermo, M. Sam, C. Silvan, V. Zaccari and R. Zafalo. Branch Prediction Techniques for Low-Power VLIW Processors. In *13th ACM Great Lakes symposium on VLSI*, pp. 225–228. 2003. ISBN 1-58113-677-3.

[77] S. Pan, K. So and J. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76–84. ACM, New York, NY, USA, October 1992.

[78] P. Panda, N. Dutt and A. Nicolau. Architectural Exploration and Optimization of Local Memory and Embedded Systems. In *International Symposium on System Synthesis*, pp. 90–97. IEEE, Washington D.C., USA, September 1997.

[79] A. Parikh, M. Kandemir and N. Vijaykrishnan. Energy Aware Instruction Scheduling. In *International Conference on High Performance Computing*, pp. 335–344. Springer, London, UK, December 2000.

[80] A. Parikh, S. Kim, M. Kankemir, N. Vijaykrishnan and M. Irwin. Instruction Scheduling for Low Power. *Journal of VLSI Signal Processing*, vol. 37(1):pp. 129–149, 2004.

[81] D. Parikh, K. Skadron, Y. Zhang, M. Barcella and M. R. Stan. Power Issues Related to Branch Prediction. In *IEE High Performance Computer Architecture*, pp. 233–244. Februrary 2002.

[82] D. Parikh, K. Skadron, Y. Zhang and M. Stan. Power-Aware Branch Prediction: Characterization and Design. *IEEE Transactions On Computers*, vol. 53(2):pp. 168 – 186, February 2004.

[83] H. Patil and J. Emer. Combining Static and Dynamic Branch Prediction to Avoid Destructive Aliasing. In *Sixth International Symposium on Computer Architecture*, pp. 251–261. IEEE, Washington D.C., USA, January 2000.

[84] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware Software Interface*. Morgan Kaufmann, Burlington, MA 01803, USA, third edn., 2005. ISBN 1558606041.

[85] C. Perleberg and J. Smith. Branch Target Buffer Design and Optimisation. *IEEE Transactions on Computers*, vol. 4:pp. 396–411, 1993.

[86] M. Predko. *Digital Electronics Demystified*. McGraw-Hill, New York, 2005.

[87] K. Roy and M. Johnson. Software Design for Low Power. In *Low Power Design in Deep Sub-Micron Design*, pp. 433–459. Kluwer Academic Press, October 1996.

[88] J. Russel and M. Jacome. Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processor. In *ICCD*, p. 328. IEEE, Washington D.C., USA, October 1998.

[89] S. Sandeep. Process Tracing Using Ptrace. *Linux Gazette*, vol. 81, 2002.

[90] T. Sato. Evaluation of Architecture-Level Power Estimation for CMOS RISC Processors. In *IEEE Symposium on Low Power Electronics*, pp. 44–45. IEEE, Washington D.C., USA, October 1995.

[91] J. S. Seng and D. M. Tullsen. Exploring the Potential of Architecture-Level Power Optimizations. In *PACS*, pp. 132–147. Springer, Berlin, Germany, 2003.

[92] W. Shiue and C. Chakrabarti. Memory Exploration for Low Power Embedded Systems. In *DAC*, pp. 250–253. IEEE, Washington D.C., USA, 1999.

[93] K. Skadron. *Characterizing and Removing Branch Mispredictions*. Ph.D. thesis, Princeton University, June 1999.

[94] K. Skadron, D. Clark and M. Martonosi. Speculative Updates of Local and Global Branch History: A Quantitative Analysis. *Journal of Instruction Level Parallelism*, vol. 2:p. 33, January 2000.

[95] K. Skadron, M. Martonosi and D. Clark. A Taxonomy of Branch Mispredictions, And Alloyed Prediction as a Robut Solution to Wrong-History Mispredictions. In *International Conference of Parallel Architectures and Compilation Techniques*, pp. 199–206. IEEE, Washington D. C., USA, October 2000.

[96] SPEC. SPEC CPU2000 Benchmarks, 2000.
URL `http://www.spec.org/cpu/`

[97] F. Steven. An Introduction to the Hatfield Super Scalar Scheduler. Tech. Rep. 316, University of Hertfordshire, June 1998.

[98] G. Steven, D. Christianson, R. Collins, R. Potter and F. Steven. A Superscalar Architecture to Exploit Instruction Level Parallelism. *Microprocessors and Microsystems*, vol. 20(7):pp. 391–400, 1997.

[99] C. Su and C. Tsui. Low Power Architecture Design and Compilation Techniques for High-Performance Processors. In *IEEE COMCON*, pp. 489–498. 1994.

[100] V. Tiwari, S. Malik and A. Wolfe. Compilation Techniques for Low Energy: An Overview. In *IEEE Symposium on Low Power Electronics*, pp. 38–39. IEEE, Washington D.C., USA, October 1994.

[101] V. Tiwari, S. Malik and A. Wolfe. A First Step Towards Software Power Minimization. In *IEEE Transactions in VLSI Systems*, pp. 437–455. December 1994.

[102] V. Tiwari, S. Malik and A. Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. In *IEEE Transactions in VLSI Systems*, vol. 2, pp. 437–445. December 1994.

[103] V. Tiwari, S. Malik and A. Wolfe. Instruction Level Power Analysis and Optimization Software. *Journal of VLSI Signal Processing Systems*, vol. 13(2), 1996.

[104] M. Toburen, T. Conte and M. Reilly. Instruction Scheduling for Low Power Dissipation in High Performance Architectures. In *Power Driven Micro-Architecture Workshop in Conjunction with ISCA*, p. 10. ACM, New York, NY, USA, June 1998.

[105] D. M. Tullsen. *Complete Guide to Semiconductor Devices*. Wiley-IEEE Press, 2 edn., 1996.

[106] Various. The Linux Kernel Archives. Online, 2005.
URL `http://www.kernel.org`

[107] Various. *Linux Operating System Manual Pages*, 2005.

[108] Various. Tux.org Discussion Lists. Online, 2005.
URL `http://www.tux.org/forums [May 2006]`

[109] L. Vintan. Towards a High Performance Neural Branch Predictor. In *The International Joint Conference on Neural Networks*, vol. 2, pp. 868–867. IEEE, Washington D. C., USA, 1999.

[110] L. Vintan, A. Gellert, A. Florea, M. Oancea and C. Egan. Understanding Prediction Limits Through Unbiased Branches. In *Advances In Computer Systems Architecture*, vol. 4186-0480 of *Lecture Notes In Computer Science*, pp. 480–487. Springer-Verlag, September 2006. ISBN 0302-9743.

[111] S. Wilton and N. Jouppi. Cacti: An Enhanced Cache Access and Cycle Time Model. *IEE Journal on Solid State Circuits*, vol. 31(5):pp. 677–688, 1996.

[112] T. Yeh and Y. Patt. Two-Level Adaptive Training Branch Prediction. In *24th Annual International Symposium on Microarchitecture*, pp. 51–61. ACM, New York, NY, USA, November 1991.

[113] Y. Zhang, X. Hu and D. Chen. Global Register Allocation for Minimizing Energy Consumption. In *International Symposium on Low Power Electronics and Design*, pp. 100–102. IEEE, Washington D.C., USA, August 1999.

[114] Z. Zhu and X. Zhang. Access-Mode Restrictions for Low-Power Cache Design. *IEEE Micro*, vol. 22(2):pp. 58–71, March-April 2002.

[115] V. Zyuban. *Inherently Lower-Power High-Performance SuperScalar Architectures*. Ph.D. thesis, University of Nore Dame, Indiana, March 2000.

[116] V. Zyuban, P. G. Emma, D. Brooks, V. Srinivasan, M. Gschwind, P. Bose and P. N. Strenski. Integrated Analysis of Power and Performance of Pipelined Microprocessors. *IEEE Transactions on Computers*, vol. 53(8), August 2004.

[117] V. Zyuban and P. Kogge. Optimization of High-Performance Superscalar Architectures for Energy Efficiency. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pp. 84–89. ACM, New York, NY, USA, 2000. ISBN 1-58113-190-9.

[]

Glossary

V_{dd}	Voltage Drain, 7
V_{ss}	Voltage Source, 7
ABBM	Adaptive Branch Bias Measurement. The process of using profiling data to compare the bias of a branch to its prediction accuracy by a dynamic predictor, 40
Activity Factor	The factor in power consumption determined by the activity of the circuit, 11
Branch Prediction	The methods utilised to predict the outcome of a control flow instruction (a branch), 13
CMOS	Complimentary Metal-Oxide-Semiconductor. A technology process used to create most modern processors from MOSEFETs, 7
Combined Algorithm	An approach which aims to represent dynamic branch behaviour statically using the ABBM and use local delay region scheduling in order to reduce the number of accesses made to a dynamic branch predictor, 36
CPU	Central Processing Unit. The main processor inside a computer that is used to execute programs, 7
Delay	The amount of time a particular operation or event takes to complete, 12
Dynamic Dissipation	A loss of power in a circuit that occurs during switching events, 7
Frequency Scaling	The process of reducing the clock frequency of a processor during periods of lower activity, 12

Gating The process of disconnecting either the clock signal or power source from an idle unit, 10

Hint-Bits Used to pass information from the compiler to the processor in the form of additional bits in an instruction, 42

Local Delay Region Scheduling The process of moving branch independent instructions into the n-slot delay region of a branch, 36

Nanometre Denotes 10^{-9} and also a technology feature size when qualified with a number, 7

Profiling The process of monitoring a program during execution in order to produce statistics, 38

Static Dissipation A constant loss of power in a circuit that occurs irrespective of switching activity, 7

Transistor Threshold The voltage level required on the gate terminal of a transistor in order to allow a current to flow from V_{dd} to V_{ss}, 7

VLSI Very Large Scale Integration. The term commonly used to describe integrated circuits consisting of thousands or more transistors, 7

Appendix A: Published Papers

Towards an Energy Efficient Branch Prediction Scheme Using Profiling, Adaptive Bias Measurement and Delay Region Scheduling

Michael Hicks, Colin Egan, Bruce Christianson, Patrick Quick
Compiler Technology and Computer Architecture Group (CTCA)
University of Hertfordshire, College Lane, Hatfield, AL10 9AB, UK
E-Mail: m.hicks@herts.ac.uk, c.egan@herts.ac.uk

***Abstract*—Dynamic branch predictors account for between 10% and 40% of a processor's dynamic power consumption. This power cost is proportional to the number of accesses made to that dynamic predictor during a program's execution. In this paper we propose the combined use of local delay region scheduling and profiling with an original adaptive branch bias measurement. The adaptive branch bias measurement takes note of the dynamic predictor's accuracy for a given branch and decides whether or not to assign a static prediction for that branch. The static prediction and local delay region scheduling information is represented as two hint bits in branch instructions. We show that, with the combined use of these two methods, the number of dynamic branch predictor accesses/updates can be reduced by up to 62%. The associated average power saving is very encouraging; for the example high-performance embedded architecture n average global processor power saving of 6.22% is achieved.**

Keywords: Branch Prediction, Power Consumption, Biased Branches, Profiling.

I. Introduction

The latency associated with branch instructions can be overcome by various means; these include branch prediction (dynamic and static), hardware multithreading, delayed branches and branch removal by aggressive static instruction scheduling. Currently, state-of-the-art processors tend to use dynamic branch prediction, but the use of dynamic predictors can consume large amounts of the silicon space and they can also consume large amounts of the power budget. In current processors, a branch predictor can consume between 10% and 40% of the overall CPU power budget [1]. The power cost is directly proportional to the number of accesses made to the dynamic predictor [2] and the power cost of modern dynamic branch predictors is comparable to that of a cache. In high performance architectures such costs may be acceptable, but we argue that such profligate use of silicon area is unlikely to be cost effective in low-power applications and will be an unnecessary drain on power. The effective use of silicon space and low power consumption is crucial for the embedded processor market. With the increasing pipeline depth of embedded processors, the accuracy of branch prediction is now becoming an important factor for embedded processor performance and a mispredicted branch will severely impact on performance. At the same time, power consumption must be kept to a minimum to ensure device usage longevity.

There is an equally valid counter argument that considers the power cost of a highly accurate dynamic predictor is offset by the effects of its accuracy [2]. Even though a dynamic predictor uses a great deal of power, the increased prediction accuracy and improved processor performance it provides results in power saving by the reduced number of branch mispredictions, negating the necessity of stalling the processor. This is because considerable power is consumed in a branch misprediction by the execution of instructions that cannot be allowed to be committed and recovery to some safe state.

In this paper, we present an approach for reducing the number of accesses and updates made to a dynamic branch predictor that combines scheduling the local delay region with profiling using an adaptive bias measurement. In the latter half of this paper, encouraging experimental results are presented. The results detail the extent to which the number of dynamic branch predictor accesses can be reduced, and also the amount of power that can be saved in an example architecture.

II. Predicting Biased Branches

In many cases the direction of a branch tends to be biased to either the taken path or to the not-taken path and therefore demonstrates a skewed distribution, which is alternatively referred to as bimodal. Using profiling, the frequency of executed branch paths can be determined and then used as a basis for predicting future runs of the program [4] [5]. In profile based prediction methodologies a static prediction-bit, or hint bit, is usually incorporated into the branch instruction format. This bit is used by the compiler to specify the prediction direction based on the branch's bias (taken or not-taken). Since they use the observed dynamic behaviour of branches, profile based branch prediction schemes differ from other static branch prediction schemes that use compile time heuristics. Also unlike dynamic branch prediction schemes, static profiling does not require large amounts of hardware support.

In this paper we use statically profiled branch prediction in conjunction with a dynamic predictor. The idea of statically profiled branch prediction is to avoid accessing the dynamic predictor whenever possible, thereby saving power. However, the profiled static prediction must be accurate to ensure that there is no impact on dynamic prediction accuracy, since inaccurate predictions are expensive in terms of both performance and power [6] [2] [7]. In the static code the number of biased branches appears to be small, but during program execution biased branches tend to be executed repeatedly and are therefore executed frequently [8]. Dynamic prediction is unnecessary for such biased branches as the direction of the branch can be statically profiled. This approach will reduce the number of accesses to the dynamic predictor and therefore save power.

Using some form of hint bits, a profiled static prediction can be used either to bypass the dynamic predictor or be used as a fallback when no prediction information is available dynamically. To save power in embedded processors it is desirable to remove dynamic predictions whenever possible. The drawback to removing biased branches from dynamic prediction using traditional compiler loop analysis is that any static prediction reflected by this method will almost always be less accurate than a dynamic prediction.

This drawback is intertwined with the definition of a biased branch. Previous approaches have used a fixed bias level [9], or, in effect, no particular bias level at all; a branch is simply marked as "likely to be taken" or "unlikely to be taken". Scant regard is given to how this will reconcile with the behaviour of the dynamic predictor in which it will be executing, and often the dynamic predictor will be more accurate [10]. Consequently, branch removal in this way impacts on performance and increases power consumption.

In our approach we take such problems into account and we propose the use of the dynamic bias measurement technique with profiling and local delay region scheduling. In local delay region scheduling the compiler schedules instructions from the same basic-block into the delay region following a branch instruction. These instructions, are those that would be executed irrespective of branch outcome and such that program semantics remain unaffected.

III. Adaptive Bias Measurement Through Profiling

Removing branches statically has traditionally been used either as an alternative to branch prediction, or, when used in conjunction with a dynamic predictor, it has been used as a fallback mechanism. The limitation of profiled static bias prediction from compiler branch heuristic analysis is accuracy. An improvement over simple heuristic static code analysis is to profile the compiled program at runtime to monitor how it behaves.

A. Profiling

Profiling, in this case, refers to the observation of a given program, at the assembly/machine level, while undergoing execution with a sample dataset [4] [5]. This means each branch instruction can be monitored in the form of a program trace by a detailed history of selected instructions and any relevant information extracted and used to form profiled static predictions where possible. A profiler is any application/system which can produce such data by observing a running program. The number of datasets that any given program is profiled with will affect the likely 'real' accuracy of the profiling results. Using a diverse range of datasets means the results will be more widely applicable.

The advantage of profiling over heuristic static code analysis is that a more precise boundary, or bias percentage, can be set for what constitutes a branch as heavily biased, and hence could be removed from dynamic prediction. Profiled traces permit the exact bias of a branch instruction to be known resulting in higher prediction accuracies.

B. Adaptive Bias Measurement

Profiling will be out performed by dynamic prediction for many branches unless the bias level is set so high that only extremely biased branches are removed and therefore profiling should be used with due caution. Consequently, we only only assign a profiled prediction to a branch where avoiding dynamic prediction has no significant negative impact on that branch's dynamic prediction accuracy.

When profiling each branch in a program's execution, an ideal profiler records the directional history for each branch, and also the prediction history [10] [5]. From this record or trace, we compute whether a branch's bias is equal to, or greater than its associated prediction accuracy from the dynamic predictor. This computation is key to the results we present in this paper. Assuming a program is profiled against an adequately varied data set, we show that these branches can safely be removed from dynamic prediction through the use of profiled hint bits. This approach to bias measurement also has a beneficial side-effect that it removes a significant number of difficult-to-predict branches. Furthermore, a branch with a very low prediction accuracy, but a higher bias will be caught by this method. Difficult-to-predict branches have a significant impact on both performance and power consumption [6] [8].

IV. Local Delay Region

Local delay region scheduling is the process of scheduling branch independent instructions from before the branch in the same basic-block into the delay slots to be executed by the processor after the branch. A branch independent instruction is any instruction whose result is not directly, or indirectly, depended upon by the branch to compute its own behaviour. The locally scheduled delay region, for a given branch, is executed irrespective of the branch direction outcome, and removes the need to predict for any branch where it can be used. Figure 1 demonstrates the process.

It is not beneficial to use local delay region scheduling in well optimised code and, where the delay region is very large such as in deeply pipelined processors. This study is focused on the embedded market where the number of delay slots tend

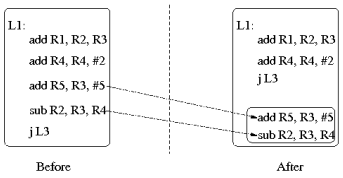

Fig. 1. Local delayed branch scheduling

to be low. Local delay region scheduling can be useful by careful use and leaving other branches untouched for dynamic prediction. Local delay region scheduling works particularly well for an unconditional absolute branch that has a fixed target address. Consequently, we propose the use of the local delay region in conjunction with adaptive bias measurement.

V. H/W Implementation

The hardware modifications required to convey information about static predictions, and therefore avoid the dynamic predictor for a given branch, are relatively simple.

Many modern processors already predecode instructions to determine whether to access the dynamic predictor unit; in which case, hint information need only be included with the branch instructions themselves. Some modern embedded instruction sets [11] already include hint bits in branch instructions, although they are only used as a fallback. We incorporate two hint bits into the branch instruction format, where the two hint bits provide profiled branch behaviour information. To our knowledge no compiler makes use of the two hint bits as we describe in this paper, which represent the following branch behaviour:

1) Statically predict taken. Do not access, or update, the dynamic predictor for this branch.
2) Statically predict not-taken. Do not access, or update, the dynamic predictor for this branch.
3) Use the locally scheduled delay region. Do not access, or update, the dynamic predictor for this branch.
4) Use the dynamic predictor.

By default all instructions would be set to case 4. The algorithm that describes how these hint bits are set is explained in the following section.

In case 1, we have hinted that the branch should always be assumed taken. This means that no access is required to the direction prediction logic in the branch predictor unit. However, the processor does not know until the decode stage what the target address will be. Rather than any complexities of computation, this is largely down to there being several different formats for branch instructions, and thus the position of the target bits in the instruction is not known. Our simulations have shown us that this is not the problem it seems at first: over 75% of dynamic branches found to be dynamically biased fall into a single instruction format - typically a single kind of offset-branch (another 18% are absolute jump instructions, but for these we use local delay region scheduling). This means that with minimal hardware modification, simple logic can be introduced to produce the target address for case 1 branches, and hence avoid accessing the Branch Target-address Cache (BTC) for predicted taken branches; this is significant for power aware designs. In the case that the hint bits provide an incorrect prediction, the existing dynamic branch prediction logic is used to recover in the same way as a dynamic misprediction (case 4).

The hardware modifications are shown in Figure 2. The block below the I-Cache represents a fetched example instruction (in this case a hinted taken branch). The simple decoder is a very small piece of hardware required to decode the two hint bits from an instruction into the relevant processor signals. The label for the local delay region is simply to indicate that this hint must be used later (in the exe pipeline stage). Additionally, outside of this diagram, during execution pipeline stage, the two hint bits must also be examined to ensure that no update occurs to the branch predictor which is also very important for power aware processors.

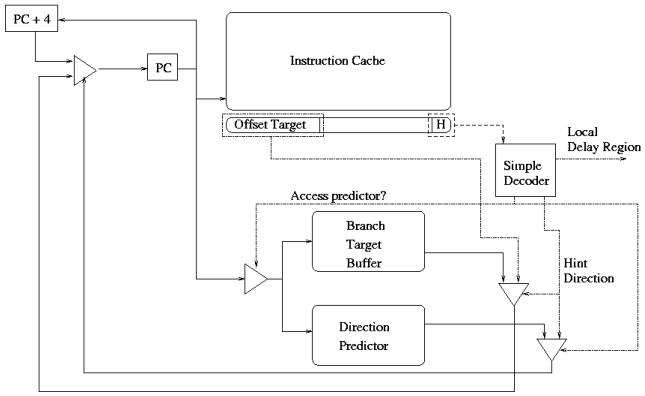

Fig. 2. A simplified block diagram representation of the hardware modifications required in the Instruction Fetch stage for the hardware simplicity approach, in order to implement the universal hint bits

VI. Simulations

This section details the process of implementation and testing of our dynamic branch prediction reduction methods, and the associated hardware modifications.

A. EEMBC and Wattch

We use the Electronic Embedded Microprocessor Benchmark Consortium (EEMBC) benchmarks [12]. EEMBC was chosen instead of the SPEC benchmarks as they represent a more appropriate target for this type of algorithm.

EEMBC [12] is a benchmark suite consisting of around forty separate benchmarks that are divided into five sections, or subsuites: Automotive, Consumer, Networking, Office and Telecom. Each subsuite represents a further specialisation towards a particular behaviour characterisation. Table I shows a simple breakdown of the five subsuites in EEMBC. Every benchmark in the suite was executed to completion to generate the results shown in the next subsection.

We use a modified version of Wattch [13], which itself is a variation of the SimpleScalar processor simulator. Wattch uses the Portable Instruction Set [Architecture] (PISA), which

TABLE I
THE FIVE EEMBC SUBSUITES WITH A LIST OF SOME OF THE TYPES OF BENCHMARKS CONTAINED WITHIN EACH SUITE

SubSuite	Benchmarks (selection of largest)
Automotive	Angle-To-Time Conversion, Fast Fourier Transform, Matrix Math...
Consumer	JPEG Compression/Decompression, RGB to CMYK, Grayscale image filter...
Networking	IP Reassembly, Network Address Translation, Route Lookup...
Office	Bezier Curve Interpolation, Floyd-Stein Grayscale Dithering, Bitmap Rotation...
Telecom	Autocorrelation, Convolutional Encoder, Viterbi Decoder...

is is a variant of the Mips instruction set. The Wattch pipeline has seven stages, and two branch delay slots before branch resolution. Local delay region scheduling was used to remove dynamic predictor accesses for the unconditional absolute-jump instruction formats, and our profiled adaptive bias measurement was used to remove dynamic predictor accesses for appropriately biased offset branch format instructions. The static prediction/delay region usage information was represented using two redundant bits in the branch instructions of the PISA instruction set. The required simulated hardware modifications were minimal and were configured as described in the previous section.

B. Algorithm

The algorithm used to configure the two hint bits in each branch instruction was implemented as shown in Algorithm 1.

Input: All Assembly Files of Programs
Output: Appropriately Hinted Assembly Files
foreach *Program* **do**
  foreach *Assembly File* **do**
    foreach *Branch Instruction* **do**
      Initially, set hint bits to "Use the dynamic predictor for this branch"
      if *Branch == Unconditional Branch* **then**
        Set hint bits to use local delay region and move two instructions preceding branch into delay region (if possible)
      else
        if *Branch's Profiled Bias* $\geq$ *Dynamic Branch Predictor's Accuracy for this Branch* **then**
          Set hint bits to Predict Profiled Bias
        end
      end
    end
  end
end

Algorithm 1: Dynamic Branch Prediction Reduction Algorithm

Figure 4 3 shows our profiling and hinting mechanism. All of the runtime profiling was conducted on a 'training' input dataset. The results shown in the next subsection were produced using a different 'test' dataset to ensure that no bias to a particular dataset was represented.

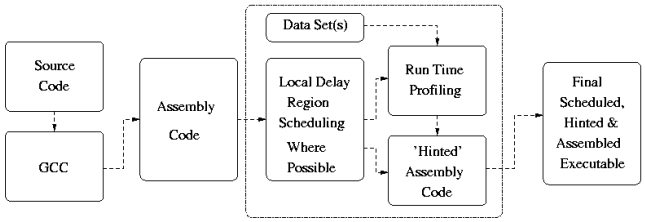

Fig. 3. Block model of the profiling and hinting regime

C. Simulation Results

The results presented and discussed in this section were produced using the processor configuration shown in Table II. Since the first part of these results is primarily observing branch instruction accuracy, the most important variable to note is branch predictor used. The GShare predictor [14] was chosen for it's accuracy, size and general applicability. The other specifications of the system were selected as a representation of a modest high-performance embedded CPU for use in applications such as PDAs and mobile telephones.

TABLE II
BASELINE CONFIGURATION USED TO GENERATE THE RESULTS SHOWN IN THIS SECTION

HWattch Parameter	Value
Issue/Decode Width	2 Instructions
Delay Slots	3
Branch Predictor	GShare
Direction Predictor Table Entries	1024
BP History Width	8Bits
BTB Sets/Associativity	512 Entries/4 Way
Data Cache Size	1024 Sets/64bit Block Size/4Way
Instruction Cache Size	512 Sets/32bit Block Size
Clock Gating Regime Modelled	Aggressive conditional clocking (non-ideal) 15% power dissipation with zero accesses
Compiler for EEMBC	gcc -O2

Table III shows the occurrence of different branch instructions across the execution of the entire EEMBC benchmark suite. The static occurrence refers to the proportion of branches accounted for by a particular branch type in the static assembly code. Dynamic occurrence refers to the proportion of branches accounted for dynamically by a particular branch type. Additionally, the third column shows whether this branch type can be removed from dynamic prediction by either of the techniques proposed. From this table we can see that the majority of dynamic branches are potential candidates for removal by either technique. Why a particular method can be used, or not, is explained in the previous section.

1) Dynamic Predictor Access Reduction: Table IV shows the success of using local delay scheduling and adaptive bias measurement to remove accesses to the branch predictor. All values shown in Table IV are averages for each subsuite. Averages were used due to the vast number of benchmarks in each suite, and also because of the high behavioural similarities of

TABLE III
STATIC AND DYNAMIC BRANCH OCCURRENCE FOR EACH PISA BRANCH TYPE, AND WHICH DYNAMIC ACCESS REMOVAL METHOD CAN BE USED

Branch Instruction	Static Occurrence	Dynamic Occurrence	Applicable Method
j	10.21%	17.31%	Local Delay Region
jal	33.95%	3.58%	Local Delay Region
jr	15.54%	3.55%	None Used
jalr	2.32%	0.04%	None Used
beq	18.18%	20.23%	Bias Profiling
bne	16.46%	50.09%	Bias Profiling
blez	1.52%	2.58%	Bias Profiling
bgtz	0.27%	1.04%	Bias Profiling
bltz	0.48%	0.39%	Bias Profiling
bgez	1.06%	1.19%	Bias Profiling

each suite. The average results were calculated by taking the total across a whole subsuite and using it as the divisor for the sum of the measured variable across the whole subsuite. For instance, 'Static Hint Rate' was calculated by summing all statically hinted branches across an entire subsuite, and dividing by the entire subsuite sum of branches – not by unweighted averaging. Figure 4 represents the same results in graphical form with the addition of an overall average.

TABLE IV
RESULTS OF LOCAL DELAY REGION AND ADAPTIVE BIAS HINTING FOR EACH EEMBC SUBSUITE

SubSuite	Dynamic Branch Rate	Static Hint Rate	Dyamic Access Reduction	Dynamic Stream Change
Automotive	17.57%	22.10%	56.42%	0.92%
Consumer	17.22%	30.80%	63.53%	0.49%
Networking	24.56%	10.57%	62.50%	0.48%
Office	19.57%	30.76%	46.82%	0.1%
Telecom	12.37%	20.49%	51.71%	-0.03%
Average	18.59%	18.29%	62.01%	0.48%

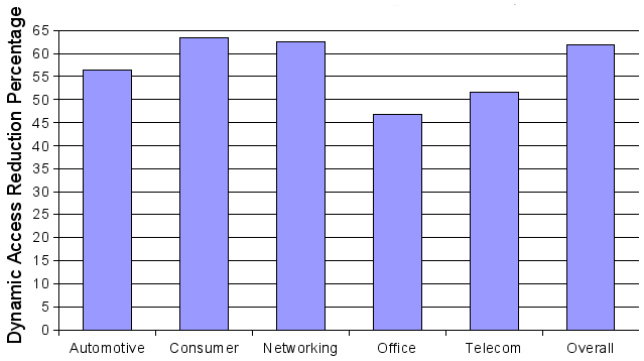

Fig. 4. The average percentage access reduction to the dynamic branch predictor for each EEMBC Subsuite

Each column in Table IV demonstrates the following:

- Dynamic Branch Rate – The percentage of instructions in the dynamic stream that were branch instructions
- Static Hint Rate – The percentage of static branches that were hinted
- Dynamic Access Reduction – The resulting reduction in accesses to the dynamic branch predictor (both the direction predictor and BTB)
- Dynamic Stream Change – The change in size (number of instructions executed) of the dynamic instruction stream as a result of the static hinting

These results demonstrate the effectiveness of the combination of local delay region scheduling and adaptive bias measurement for removing the need to dynamically predict for many branches. The overall average of 62% dynamic branch prediction reduction is extremely promising. Unsurprisingly, the Consumer subsuite performed most successfully with the algorithm. This is because of the highly cyclic nature of many of the algorithms included in this subsuite: JPEG compression and decompression for instance.

Most importantly, we can see that our dynamic branch prediction reduction algorithm has no significant detrimental effects on the performance of the program: Table IV shows that the size of the dynamic instruction stream was not significantly expanded with additional instructions from increased misprediction. In fact, in many individual cases, the number of instructions executed was reduced. This is likely accounted for by the removal of poorly dynamically predicted branches; a branch with a bias greater than its accuracy is automatically removed from dynamic prediction.

Although the average results in Table IV are representative, there were some intrasubsuite exceptions. Notably these were: Angle-To-Time Conversion benchmark in Automotive and the Viterbi Decoder benchmark in Telecom. These had dynamic prediction removal percentages of 11% and 35%, respectively.

2) Subsequent Power Saving: After simulating the number of dynamic branch predictor accesses that could be removed for the predictor used in our example architecture we then used our variant of Wattch to model the amount of power than can be saved. The dynamic branch predictor accounts for between 10% and 15% of global power dissipation in the example architecture when all branches use dynamic prediction.

Demonstrating how much power can be saved is indicative only for the example architecture used. The amount of power saved in the branch predictor itself is generally proportional to the dynamic access reduction as a result of the application of our algorithm. However, the power saved in the branch predictor gives no indication of the global power saved over the whole processor, and also does not take into account any additional delay incurred by the use of the access reduction algorithm. Providing global processor power results is thus useful, but the results depend on the relative size of the rest of the processor compared to the dynamic branch predictor unit.

The average global processor power savings per committed instruction, for the architecture in Table II, are shown in Table V. The results per committed instruction were calculated by dividing the total global power consumed during a program's execution by the number of committed instructions. The power saving per committed instruction implicitly takes into account any change in the size/delay of the instruction stream as the number of committed instructions remains the

same for all test executions of the benchmarks; an increase of the number of instructions executed after the application of the reduction algorithm would scale the power saved in the branch predictor when calculated for the committed instructions even though branch predictor accesses may have been reduced.

TABLE V
AVERAGE POWER SAVING PER COMMITTED INSTRUCTION FOR NON-IDEAL CLOCK GATING REGIME AND *IDEAL CLOCK GATING REGIME

EEMBC Subsuite	Average Power Saving	Best/Worst
Automotive	5.43% (*12.38%)	14.87% / 10.15%
Consumer	6.17% (*12.69%)	10.66% / 9.73%
Networking	6.84% (*14.53%)	14.73% / 10.60%
Office	5.66% (*13.56%)	11.38% / 10.07%
Telecom	4.10% (*10.07%)	11.14% / 9.21%
Overall	6.22% (*13.47%)	N/A

The standard power saving results in Table V are for the non-ideal clock gating regime described in the processor specification Table II. However, we have additionally included the power saving results for a more ideal clock gating algorithm with close to zero dissipation on zero accesses. These additional results are denoted with an asterisk (*).

Figure 5 shows the standard, non-ideal results, but also includes the average power saving per instruction as if no accesses were made to the dynamic branch predictor (Free BP – whilst still maintaining the same prediction accuracy). This allows a comparison between the success of the power saving and the absolute ceiling value possible.

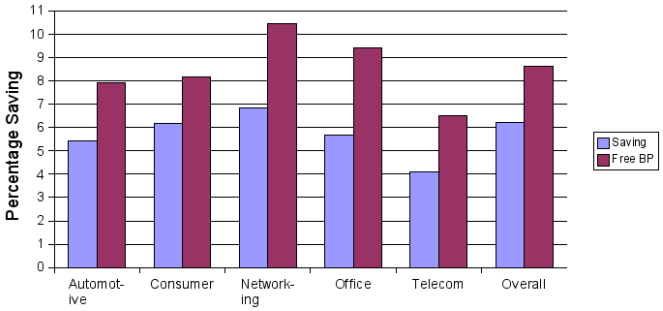

Fig. 5. Average power saving per committed instruction

Although Figure 5 shows that the algorithm is not 'perfect', we must remember that not all branch prediction accesses are removed and as such it will not be possible to be ideal without impacting heavily on performance, and thus power.

VII. CONCLUSIONS AND FUTURE WORK

Dynamic branch predictors cannot be removed from processors while high performance and low power consumption are issues [6]. However, the results in this paper have shown that, in an embedded context, the number of accesses that need to be made throughout a program's execution can be dramatically reduced: in this example architecture by 62%. While previous attempts at power saving have focused on the introduction of hardware units to monitor dynamic behaviour, our approach can achieve similar levels of access reduction, but without the need to significantly modify hardware. The number of dynamic predictor accesses that can be removed with this approach is highly dependent on the accuracy of the dynamic predictor being used. In this paper we used a very accurate dynamic predictor, but a higher reduction can be achieved in architectures with a less accurate dynamic branch predictor. Additionally, this approach is applicable to both SuperScalar and VLIW processors.

The amount of power saved, for the non-ideal clock gating regime, averaged at 6.22% global power saving across the EEMBC benchmark suites. This result is significant and very encouraging; in architectures where the branch predictor is relatively more expensive (in terms of power) this figure will be higher. When considering the predecode logic that already exists in most processors, the hardware modifications are minor and easy to accommodate. The time required to simulate, profile, assign static predictions and generate results was under 90 minutes for the entire EEMBC suite on a standard modern desktop machine, and this process is required only once before a program's distribution. For these small costs, an average power saving of at least 6% is highly attractive for processors in embedded devices that account for a large proportion of the whole device's limited energy budget.

REFERENCES

[1] Parikh, D., Skadron, K., Zhang, Y., Barcella, M., Stan, M.R.: Power issues related to branch prediction, IEEE HPCA (2002)
[2] Egan, C., Hicks, M., Christianson, B., Quick, P.: Enhancing the i-cache to reduce the power consumption of dynamic branch predictors, IEEE Digital System Design (July 2005)
[3] Seng, J.S., Tullsen, D.M.: Exploring the potential of architecture-level power optimizations, PACS (2003)
[4] Fisher, J.A., Freudenberger, S.: Predicting conditional branch directions from previous runs of a program. In: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Boston. (October 1992)
[5] Hicks, M., Egan, C., Christianson, B., Quick, P.: Htracer: A dynamic instruction stream research tool, IEEE Digital System Design (July 2005)
[6] Parikh, D., Skadron, K., Zhang, Y., Stan, M.: Power aware branch prediction: Characterization and design. IEEE Transactions On Computers **53**(2) (February 2004)
[7] Martin, A.J., Nystrom, M., Penzes, P.L.: Et^2: A metric for time and energy efficiency of computation. (2003)
[8] Vintan, L., Gellert, A., Florea, A., Oancea, M., Egan, C.: Understanding prediction limits through unbiased branches. In: Advances In Computer Systems Architecture. Volume 4186-0480 of Lecture Notes In Computer Science., Springer-Verlag (September 2006) 480487
[9] Jacobsen, E., Rotenberg, E., Smith, J.: Assigning confidence to conditional branch predictions, IEEE 29th International Symposium on Microarchitecture (1996)
[10] Hicks, M., Egan, C., Christianson, B., Quick, P.: Reducing the branch power cost in embedded processors through static scheduling, profiling and superblock formation. In: Advances In Computer Systems Architecture. (September 2006)
[11] IBM: PowerPC Instruction Set Manual. (2005)
[12] Levy, M.: The embedded microprocessor benchmark consortium. Online (2005)
[13] Brooks, D., Tiwari, V., Martonosi, M.: Wattch: a framework for architectural-level power analysis and optimizations, 27th annual international symposium on Computer architecture (2000)
[14] Egan, C.: Dynamic Branch Prediction In High Performance Super Scalar Processors. PhD thesis, University of Hertfordshire (August 2000)

Reducing the Branch Power Cost In Embedded Processors Through Static Scheduling, Profiling and SuperBlock Formation

Michael Hicks, Colin Egan, Bruce Christianson, Patrick Quick

Compiler Technology and Computer Architecture Group (CTCA)
University of Hertfordshire, College Lane, Hatfield, AL10 9AB, UK
m.hicks@herts.ac.uk

Abstract. Dynamic branch predictor logic alone accounts for approximately 10% of total processor power dissipation. Recent research indicates that the power cost of a large dynamic branch predictor is offset by the power savings created by its increased accuracy. We describe a method of reducing dynamic predictor power dissipation without degrading prediction accuracy by using a combination of local delay region scheduling and run time profiling of branches. Feedback into the static code is achieved with hint bits and avoids the need for dynamic prediction for some individual branches. This method requires only minimal hardware modifications and coexists with a dynamic predictor.

1 Introduction

Accurate branch prediction is extremely important in modern pipelined and MII microprocessors [10] [2]. Branch prediction reduces the amount of time spent executing a program by forecasting the likely direction of branch assembly instructions. Mispredicting a branch direction wastes both time and power, by executing instructions in the pipeline which will not be committed. Research [8] [3] has shown that, even with their increased power cost, modern larger predictors actually save global power by the effects of their increased accuracy. This means that any attempt to reduce the power consumption of a dynamic predictor must not come at the cost of decreased accuracy; a holistic attitude to processor power consumption must be employed [7][9].

In this paper we explore the use of delay region scheduling, branch profiling and hint bits (in conjunction with a dynamic predictor) in order to reduce the branch power cost for mobile devices, without reducing accuracy.

2 Branch Delay Region Scheduling

The branch delay region is the period of processor cycles proceeding a branch instruction in the processor pipeline before branch resolution occurs. Instructions can fill this gap either speculatively, using branch prediction, or by the use of scheduling. The examples in this section use a 5 stage MIPS pipeline with 2 delay slots.

2.1 Local Delayed Branch

In contrast to scheduling into the delay region from a target/fallthrough path of a branch, a locally scheduled delay region consists of branch independent instructions that precede the branch (see Figure 1). A branch independent instruction is any instruction whose result is not directly or indirectly depended upon by the branch to calculate its own behaviour.

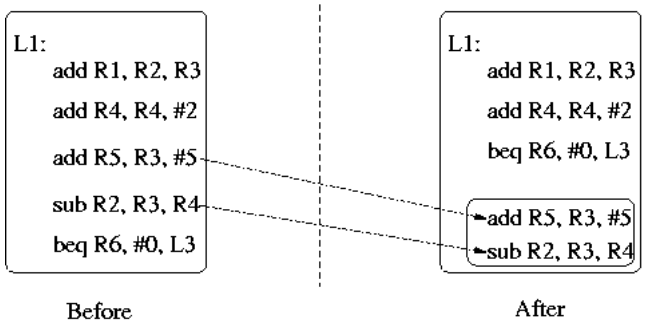

Fig. 1. An example of local delayed branch scheduling.

Deciding which instructions can be moved into the delay region locally is straightforward. Starting with the instruction from the bottom of the given basic block in the static stream, above the branch, examine the target register operand. If this target register is NOT used as an operand in the computation of the branch instruction then it can be safely moved into the delay region. This process continues with the next instruction up from the branch in the static stream, with the difference that this time the scheduler must decide whether the target of the instruction is used by any of the other instructions below it (which are in turn used to compute the branch).

Local Delay Region Scheduling is an excellent method for utilising the delay region where possible; it is always a win and completely avoids the use of a branch predictor for the given branch. The clear disadvantage with local delay region scheduling is that it cannot always be used. There are two situations that result in this: well optimised code and deeply pipelined processors (where the delay region is very large). It is our position that, as part of the combined approach described in this paper, the local delay region is profitable.

3 Profiling

Suppose that we wish to associate a reliable static prediction with as many branches as possible, so as to reduce accesses to the dynamic branch predictor of a processor at runtime (in order to save power). This can be achieved to a reasonable degree through static analysis of the assembly code of a program; it is often clear that branches in loops will commonly be taken and internal break points not-taken.

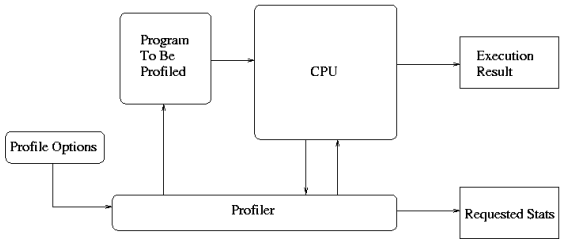

Fig. 2. The profiler is supplied with parameters for the program and the traces/statistics to be logged

A more reliable method is to observe the behaviour of a given program while undergoing execution with a sample dataset [4]. Each branch instruction can be monitored in the form of a program trace and any relevant information extracted and used to form static predictions where possible. A profiler is any application/system which can produce such data by observing a running program (see Figure 2). The proceeding two sections examine the possibility of removing certain classes of branch from dynamic prediction by the use of run-time profiling.

3.1 Biased Branches

One class of branches that can be removed from dynamic prediction, without impacting on accuracy, are highly biased branches. A biased branch is a branch which is commonly taken or not taken, many times in succession before possibly changing direction briefly. The branch has a bias to one behaviour. These kinds of branches can, in many cases, be seen to waste energy in the predictor since their predicted behaviour will be almost constantly the same [5] [8].

The principles of spatial and temporal locality intuitively tell us that biased branches account for a large proportion of the dynamic instruction stream. Identifying these branches in the static code and flagging them with an accurate static prediction would enable them to be executed without accessing the dynamic predictor. The profiler needs to read the static assembly code and log, for each each branch instruction during profiling, whether it was taken or not taken at each occurrence.

3.2 Difficult to Predict Branches (Anti Prediction)

Another class of branch instructions that would be useful to remove from dynamic branch predictor accesses are difficult to predict branches. In any static program there are branches which are difficult to predict and which are inherently data driven. When a prediction for a given branch is nearly always likely to be wrong, there is little point in consuming power to produce a prediction for it since a number of stalls will likely be incurred anyway [5] [8] [6].

Using profiling, it is possible to locate these branches at runtime using different data sets and by monitoring every branch. The accuracy of each dynamic prediction is required rather than just a given branch's behaviour. For every

branch, the profiler needs to compare the predicted behaviour of the branch with the actual behaviour. In the case of those branch instructions where accuracy of the dynamic predictor is consistently poor, it is beneficial to flag the static branch as difficult to predict and avoid accessing the branch predictor at all, letting the processor assume the fallthrough path. Accordingly, filling the delay region with NOP instructions wastes significantly less power executing instructions that are unlikely to be committed.

4 Combined Approach Using Hint Bits

The main goal of the profiling techniques discussed previously can only be realised if there is a way of storing the results in the static code of a program, which can then be used dynamically by the processor to avoid accessing the branch prediction hardware [3].

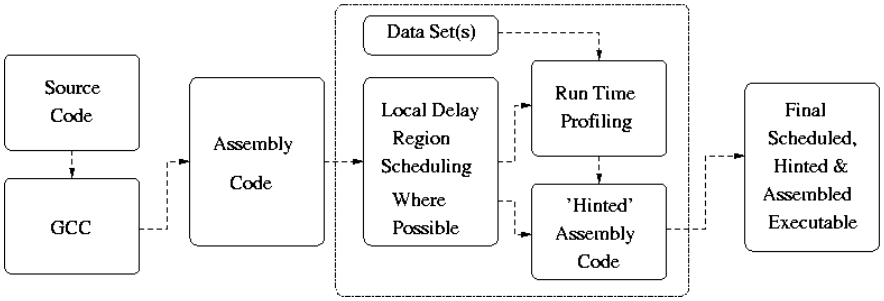

Fig. 3. Block diagram of the proposed scheduling and hinting algorithm. The dotted box indicates the new stages introduced by the algorithm into the creation of an executable program

The combined approach works as follows:

1. Compile the program, using GCC for instance, into assembly code.
2. The Scheduler parses the assembly code and decides for which branch instructions the local delay region can be used (see section 2.1).
3. The Profiler assembles a temporary version of the program and executes it using the specified data set(s). The behaviour of each branch instruction is logged (see section 3).
4. The output from the profiling stage is used to annotate the delay scheduled assembly code.
5. Finally, the resulting annotated assembly code is compiled and linked to form the new executable.

The exact number of branches that can be eliminated from runtime predictor access in the target program depends upon the tuning of the profiler and the number of branches where the local delay region can be used.

4.1 Hint Bits

So far we have described a process of annotating branch instructions in the static assembly code to reflect the use of the local delay region and of the profiling results. The way this is represented in the assembly/machine code is by using an existing method known as hint bits (though now with the new function of power saving).

The four mutually exclusive behaviour hints in our algorithm which need to be stored are:

1. Access the branch predictor for this instruction.
2. or Assume this branch is taken (don't access dynamic predictor logic).
3. or Assume this branch is not taken (don't access dynamic predictor logic).
4. or Use this branch's local delay region (don't access dynamic predictor logic).

The implementation of this method requires two additional bits in an instruction. Whether these bits are located in all of the instruction set or just branches is discussed in the proceeding section. Another salient point is that the information in a statically predicted taken branch replaces only the dynamic direction predictor in full; the target of the assumed taken branch is still required. Accessing the Branch Target Buffer is costly, in terms of power, and must be avoided.

Most embedded architectures are Reduced Instruction Set Computers [8]. Part of the benefit of this is the simplicity of the instruction format. Since most embedded system are executing relatively small programs, many of the frequently iterating loops (the highly biased branches, covered by the case 2 hint) will be PC relative branches. This means that the target address for a majority of these branches will be contained within a fixed position inside the format. This does not require that the instruction undergo any complex predecoding, only that it is offset from the current PC value to provide the target address. Branch instructions that have been marked by the profiler as having a heavy bias towards a taken path, but which do not fall into the PC relative fixed target position category have to be ignored and left for dynamic prediction.

The general 'hinting' algorithm:

1. Initially, set the hint bits of all instructions to: assume not taken (and do not access predictor).
2. Set hint bits to reflect use of the local delay region where the scheduler has used this method.
3. From profiling results, set hint bits to reflect taken biased branches where possible.
4. All remaining branch instructions have their hint bits set to use the dynamic predictor.

4.2 Hardware Requirements/Modifications

The two possible implementation strategies are:

Hardware Simplicity: Annotate every instruction with two hint bits. This is easy to implement in hardware and introduces little additional control logic. All non branch instructions will also be eliminated from branch predictor accesses. The disadvantages of this method are that it requires that the processor's clock frequency is low enough to permit an I-Cache access and branch predictor access in series in one cycle and that there are enough redundant bits in all instructions.

Hardware Complexity: Annotate only branch instructions with hint bits and use a hardware mechanism similar to a Prediction Probe Detector [8] to interpret hint bits. This has minimal effect on the instruction set. It also means there is no restriction to series access of the I-Cache then branch predictor. The main disadvantage is the newly introduced PPD and the need for instructions to pass through the pipeline once before the PPD will restrict predictor access.

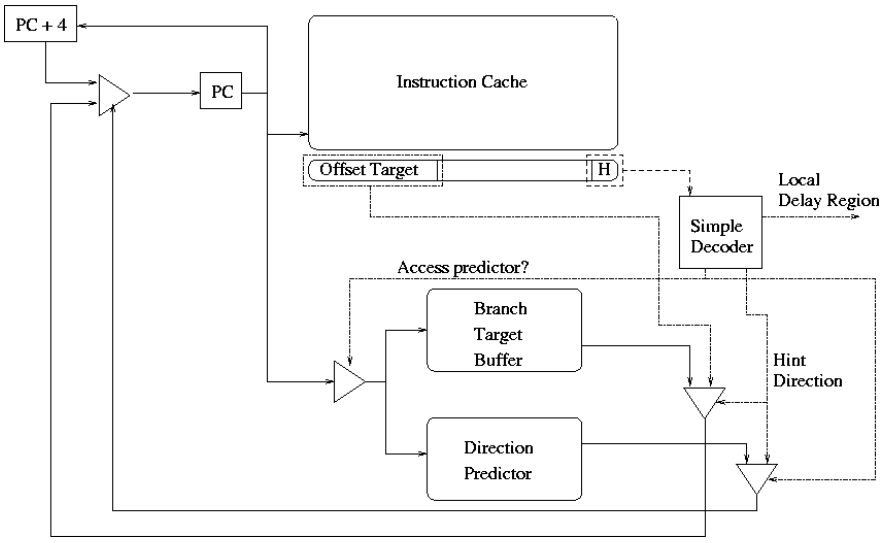

Fig. 4. Diagram of required hardware modifications. The block below the I-Cache represents a fetched example instruction (in this case a hinted taken branch).

The hardware simplicity model offers the greatest power savings and is particularly applicable for the embedded market where the clock frequency is generally relatively low, thus a series access is possible. It is for these reason we the use the hardware simplicity model. In order to save additional power, some minor modifications must be made to the Execution stage to stop the statically predicted instruction from expending power writing back their results to the predictor (since their results will never be used!).

It can be seen that after a given program has had its hint bits set, all of the branches assigned static predictions (of taken or not taken) have now essentially formed superblocks, with branch resolution acting as a possible exit point from the newly formed super block. When a hint bit prediction proves to be incorrect,

it simply acts as a new source of a branch misprediction; it is left for the existing dynamic predictor logic to resolve.

5 Conclusion and Future Work

Branch predictors in modern processors are vital for performance. Their accuracy is also a great source of powersaving, through the reduction of energy spent on misspeculation [8]. However, branch predictors themselves are often comparable to the size of a small cache and dissipate a non trivial amount of power. The work outlined in this paper will help reduce the amount of power dissipated by the predictor hardware itself, whilst not significantly affecting the prediction accuracy. We have begun implementing these modifications in the Wattch [1] power analysis framework (based on the SimpleScalar processor simulator). To test the effectiveness of the modifications and algorithm, we can have chosen to use the EEMBC benchmark suite, which provides a range of task characterisations for embedded processors.

Future investigation includes the possibility of dynamically modifying the hinted predictions contained within instructions to reflect newly dynamically discovered biased branches.

References

1. David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. 27th annual international symposium on Computer architecture, 2000.
2. Colin Egan. *Dynamic Branch Prediction In High Performance Super Scalar Processors*. PhD thesis, University of Hertfordshire, August 2000.
3. Colin Egan, Michael Hicks, Bruce Christianson, and Patrick Quick. Enhancing the I-Cache to Reduce the Power Consumption of Dynamic Branch Predictors. IEEE Digital System Design, jul 2005.
4. Michael Hicks, Colin Egan, Bruce Christianson, and Patrick Quick. HTracer: A Dynamic Instruction Stream Research Tool. IEEE Digital System Design, jul 2005.
5. Erik Jacobsen, Erik Rotenberg, and J.E. Smith. Assigning Confidence to Conditional Branch Predictions. IEEE 29th International Symposium on Microarchitecture, 1996.
6. J. Karlin, D. Stefanovic, and S. Forrest. The Triton Branch Predictor, oct 2004.
7. Alain J. Martin, Mika Nystrom, and Paul L. Penzes. ET^2: A Metric for Time and Energy Efficiency of Computation. 2003.
8. D. Parikh, K. Skadron, Y. Zhang, and M. Stan. Power Aware Branch Prediction: Characterization and Design. *IEEE Transactions On Computers*, 53(2), feb 2004.
9. Dharmesh Parikh, Kevin Skadron, Yan Zhang, Marco Barcella, and Mircea R. Stan. Power Issues Related to Branch Prediction. IEEE HPCA, 2002.
10. David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface*. Morgan Kaufmann, second edition, 1998.

HTracer: A Dynamic Instruction Stream Research Tool

Michael Hicks, Colin Egan, Bruce Christianson, Patrick Quick

Compiler Technology and Computer Architecture Group (CTCA)
University of Hertfordshire
Hatfield, Herts, U.K. AL10 9AB
m.hicks@herts.ac.uk

Modern processor design research hinges as much upon understanding the nature of the dynamic instruction stream as it does on the function and performance of the underlying hardware. With this in mind, along with the absence of a suitable existing utility, that we have created the Hatfield Tracer (HTracer) – a cross-platform tool to aid future research by extracting information about the dynamic instruction stream of real world applications/benchmarks on real world machines.

A tracer, or trace tool, allows the user to view information and 'trace' the execution of a program or of particular instructions within that program.

HTracer will monitor the execution of a specified program and dump the state of the processor either after every instruction or after those that have been specified for tracing. This enables the user to produce either a full trace of the entire dynamic instruction stream for a given program or for specific instructions of interest. The way this information is specified is such that one may request that all branch instructions, for instance, are traced and reported as a single item or separately as specific branch types. Along with this information, HTracer can also save information about the register file/instruction operands at each instruction occurrence. One way in which this is useful is that when a full trace has been conducted the results can be used as input to simulators (to investigate for instance branch prediction or caching)and avoid further (often lengthy) runtime evaluation of the traced program.

All of this is achieved by single step execution of the process being traced which thus allows HTracer to interrupt the instruction stream and examine the state of the processor after each instruction step. After each step HTracer will examine the instruction being executed and compare it to the specified list of flexible masks; if a match occurs then the mnemonic specified with the mask is recorded along with the Instruction Pointer. At the very least the results from a trace will include the Instruction Pointer and the Instruction Mnemonic match at the corresponding occurrence in the dynamic stream, but if requested it can include almost any state information about the processor. This functionality allows for the gathering of detailed information on multiple program executions with varied input data – with which we can monitor the effect of many research areas such as compiler technologies, schedulers etc on the dynamic stream and the appearance of certain instructions.

Something which is key to the usefulness of HTracer is its ability to function on many different architectures – in fact any architecture for which a modern version of the GNU/Linux Kernel is available, thus allowing for the monitoring of the performance variation of a given modification across different architectures. These architectures include, but are not limited to: x86, AMD64, PPC32/64, ARM Chips, Sparc32/64. For instance we could trace the dynamic instruction stream of some compiled embedded benchmarks (usage characterisations) from different compilers and on different platforms and then use the outputs (discussed below) to test the effects of the compilers and/or mod-

ifications on the dynamic stream.

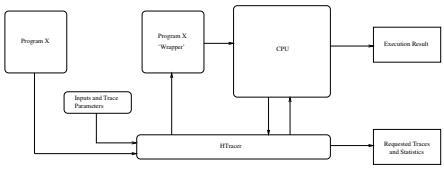

Figure 1: Logical block diagram of the HTracer tool.

Figure 1 shows in simple form how HTracer can be used: we supply it with a program to trace, the parameters to that program, an input file expressing which instructions to trace (if not all) and finally details of what state information (such as the target of branches, register file and so on) we wish to be saved at the appearance of each of these instructions in the stream.

At the completion of a trace run we are furnished with a file containing all of the results for the requested trace. These can either be hand analysed (since we can turn on tagged output – improving the readability of the file) or we can use the simply formatted results as the input to another tool. These can include such things as branch prediction simulators (as a list of verified branch addresses and computed targets), trace driven architecture simulators or perhaps instruction based power usage estimation tools. Again a key advantage here is that we can then easily input traced data from various different architectures into a single simulator – this can be achieved since, as a by-product of the instruction-to-mnemonic trace mapping, all instructions can be recorded in the same general form across different architectures. Another output which can be obtained from HTracer is statistical information about instruction occurrence in the dynamic stream.

Given the modular library design of HTracer, is it possible to either generate the results in the form of a file (as previously discussed) or as a stream which can be used dynamically as the input to other tools allowing live trace input and opcode translation.

At the University of Hertfordshire HTracer is actively being used to generate source data for all of the above simulators (most notably in my own research project, investigating the effects of instruction scheduling and branch prediction on embedded processor power consumption) and as such is under constant development. The source code for HTracer can be made available to other parties upon request to the author.

Enhancing the I-cache to Reduce the Power Consumption of Dynamic Branch Predictors

Colin Egan, Michael Hicks, Bruce Christianson and Patrick Quick

Compiler Technology and Computer Architecture Group (CTCA), University of Hertfordshire, Hatfield, Hertfordshire, U.K. AL10 9AB
c.egan@herts.ac.uk

Branch prediction is an important area of research for High Performance Processors [1]. With ever increasing depth of pipeline and an increase in the number of instructions issued in each clock cycle the penalty associated with a mispredicted branch impacts increasingly severely on processor performance [2]. Over recent years branch prediction research has focussed on increasing prediction accuracy with scant regard for cost and power [2, 3]. For example, Patt argues that between 256K and 1024K bytes of the silicon budget should be devoted to branch prediction [4, 5], but battery life expectancy in embedded processor systems (such as mobile phones) and laptops is an increasingly important usage factor. In current processors a branch predictor can consume more than 10% of the power [6]. In high performance architectures this may be an acceptable cost, but in embedded processors and laptops such large power consumption is a major factor limiting the quality and quantity of branch prediction that can be attempted [7, 8].

There are various means to overcome the problems associated with branch instructions; these include branch prediction (dynamic and static), hardware multithreading, delayed branches and branch removal by static instruction scheduling. Currently state of the art processors tend to use dynamic predictors, but such predictors can consume large amounts of the silicon and the power budget. With the development of two-level predictors in the 1990s by Yale Patt's group [9] and by Pan, So and Rahmeh [10] researchers have reported very high prediction accuracy, but this success is only achieved by providing very large arrays of prediction counters or PHTs (Pattern History Tables). We argue that such profligate use of silicon area is unlikely to be cost effective in low-power applications and will be an unnecessary drain on power. The effective use of silicon space and low power consumption is crucial for the embedded processor market. With the increasing pipeline depth of embedded processors, the accuracy of branch prediction is now becoming an important factor for embedded processor performance and a mispredicted branch will severely impact on performance. At the same time, power consumption must be kept to a minimum to ensure device longevity.

Following the principle of locality, our proposal is to enhance each I-cache block with two additional bits. The first bit would indicate whether the instruction is a branch or not and the second bit would indicate if the branch was to be predicted taken or not taken. Since an I-cache is used to store previously executed instructions that are likely to be executed again in the near future then this branch identification information would be available after the first execution. Furthermore, in dynamic predictors (two-level and a traditional BTC) the most significant bit of the appropriate two-bit up/down saturating counter is used to furnish a prediction during the IF stage of the pipeline, and that counter is updated when the branch outcome is known at resolution. Consequently, a prediction is known well in advance of the next time that particular branch instruction is executed and its prediction bit could also be stored in the I-cache. Both bits would have to be set for the branch target address cache (BTC) to be accessed, in which case the BTC would be used to simply furnish a pre-computed target address. In the case of a branch predicted as not-taken or the first time

a branch has been encountered or any other instruction then the branch predictor and the BTC would not be accessed, thereby reducing the power consumption. This means that power would only be consumed accessing a target address in the case of a predicted taken branch. We do not consider it appropriate to store the pre-computed branch target address in the I-cache as this would considerably add to the cost of the I-cache and increase its hardware complexity. As usual, power would be required at branch resolution to update the appropriate saturating counter. A small additional amount of power would also be required to update our enhanced fields in the I-cache.

It is our intention to use the well known SimpleScalar Tool set [11] in conjunction with the Wattch Power analysis tool [12] to quantify, evaluate and validate our ideas. We intend to undertake a comparative study using the branch predictors provided by the SimpleScalar tool set. We will quantify the prediction accuracies of these predictors and the amount of power they consume. We will then enhance the I-cache as described and re-evaluate the power consumption of the same predictors. We expect to see a realistic power saving with our enhanced I-cache and expect to be able to discuss our results at conference.

We also consider that a further power saving enhancement would be the addition of a third, taken-saturated, field in the I-cache. The justification behind this is that many branches are in fact in tight loops and are therefore repeatedly taken before loop exit. This means that the associated two-bit up/down field will be saturated and the field value and the prediction remain the same. So long as the branch continues to be taken, the addition of the saturating bit in the I-cache would mean there would be no need to update the prediction field at branch resolution thereby saving more power by removing unnecessary accesses to the branch predictor. Now the BTC is only accessed in the case of a taken prediction and the predictor will not be accessed in the case of a taken saturated counter field.

In all cases the BTC and the predictor must remain powered, because removing power would mean the BTC predictor would require reinitialising each time it is accessed.

References

[1] L. Hennessy and D. A. Patterson "Computer Architecture: A Quantitative Approach". Third Edition, Elsevier/Morgan Kaufmann, San Mateo, California, 2002.

[2] C. Egan, G. B. Steven and L. N. Vintan. "Cached Two-Level Adaptive Branch Predictors with Multiple Stages". *In Trends in Network and Pervasive Computing - ARCS 2002 (LNCS 2299)*, Springer-Verlag, 2002, pp. 179-191.

[3] C.Egan. "Dynamic Branch Prediction in High Performance Superscalar Processors". PhD thesis, University of Hertfordshire, U.K. 2000.

[4] D. C. Burger and J. R. Goodman. "Billion-Transistor Architectures". *IEEE Computer*, September 1997, pp. 46–49.

[5] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly and J. Stark. "One Billion Transistors, One Uniprocessor, One Chip". 1997, *Computer*, pp. 51-57.

[6] D. Parikh, K. Skadron, Y. Zhang, M. Barcella and M. R. Stan. "Power issues related to branch prediction". P*roc. Int. Conf. on High-Performance Computer Architecture*, IEEE, 2002, pp. 233-242.

[7] S. W. Chung and S. B. Park. "A Low Power Branch Predictor to Selectively Access the BTB". *In the Ninth Asia-Pacific Computer Systems Architecture Conference (ACSAC04)*, Beijing, China, September 2004.

[8] W. Shi, T. Zhang and S. Pande. "Static Techniques to Improve Power Efficiency of Branch Predictors". *In the Ninth Asia-Pacific Computer Systems Architecture Conference (ACSAC04)*, Beijing, China, September 2004.

[9] T.-Y. Yeh and Y. N. Patt. "Two-level adaptive training branch prediction". *In Proceedings of the 24th Annual International Symposium on Microarchitecture*, November 1991, pp 51–61.

[10] S.-T. Pan, K. So, and J. T. Rahmeh. "Improving the accuracy of dynamic branch prediction using branch correlation". *In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp 76–84.

[11] D. C. Burger and T. M. Austin. "The SimpleScalar tool set, version 2.0". *Computer Architecture News*, June 1997, 25(3):13–25.

[12] D. Brooks , V. Tiwari , M. Martonosi. "Wattch: a framework for architectural-level power analysis and optimizations". *ACM SIGARCH Computer Architecture News*, May 2000, v.28 n.2, pp.83-94.

Appendix B: Technical Reports

An Introduction to Power Consumption Issues in Processor Design

M.A. Hicks, C. Egan

February 2005

Abstract

The field of power aware computer engineering is becoming increasingly prominent in recent years, following both the rapid take up of mobile devices and the increasingly large and complex design of processors. This introductory paper begins by examining the impetus behind present research and then proceeds with simple background theory of power dissipation in current electronic circuit technology. Following the definition of simple antiquated ways of reducing power dissipation are more contemporary higher level metrics used to quantify power efficiency (Et^2) and a brief summary of simulation tools used to retrieve power usage information. To finish, a summary of current approaches and questions is presented.

1 Introduction

> We're projecting by 2010 there will be more than 2.5 billion wireless handheld devices capable of providing the communications functions combined with the processing power of today's high-performance PCs.
>
> – Paul Otellini, Intel

The first question one might ask is "Why be power efficient?". Power consumption/efficiency is an increasingly important issue in the field of computer engineering. We can say this because power consumption/dissipation affects so many factors in real world implementations:

Battery Life - The most obvious restriction on mobile devices is the length of time we can use that device for and the performance we can expect. Power efficient designs can extend this but, as shown later, careful consideration must be taken in assessing the relative advantages of a design.

Thermal Issues - Power dissipation results in heat. Excessive heat dissipation will affect the design of relevant cooling systems (and thus device packaging/size), reliability and precise timing. One of the most important issues on the desktop machine is packaging size and cost. On large scale servers packaging size will represent a volumetric limit on rack capacity.

Large Scale Power Consumption - This aspect often seems irrelevant when examining the desktop and small scale market but when one looks at massive arrays of machines the resulting power consumption can be quite significant.

Given the prediction quoted from Intel one can see the extreme importance of power efficient designs in the coming decades. In fact, power efficiency is a likely linchpin for new high speed processor designs.

The basic method behind reducing power consumption is to:

1. Find out where power is being dissipated most in the system.
2. Apply a suitable metric to this.
3. Try out lots of different (relevant) approaches to reduce this!

Given this approach, the following sections will attempt to offer a brief introduction to some of the principles of the area.

2 Background Theory

This section offers some low level background theory explaining how power is dissipated in silicon transistors. If you aren't interested in understanding the details of how power is dissipated you can probably skip this section. However, the contents of this section may help you understand proceeding sections more clearly [8].

2.1 Some Definitions

Before we further examine power dissipation it is important that some definitions are known the reader, particularly since many readers from a computing background may not be familiar with electronics. A basic understanding of science is assumed.

Every electronic circuit has the following properties:
(assume $t =$ observation time)

Charge (Q) - The charge at a particular point in a circuit is defined as the number of electrons passing that point in the given time period. Charge is measured in **Coulombs**. The charge, Q, at a given point is said to be one Coulomb if 6.3 million electrons pass that point during the observation period. $Q = IT$

Current (I) - The current at a given point in a circuit is defined as the charge per second. I.E. The number of electrons passing that point every second. This is in contrast to charge which is a measure of electrons, not their flow rate. We measure current in **Amperes**. $I = \frac{Q}{T}$

Electrical Energy (W) - This is the "energy" stored in an electrical form. As with all energy we measure electrical energy in **Joules**.

Voltage (V) - Voltage can be used to express one of two things. **EMF** specifies the Electro Motive Force of a supply source. **PD** (Potential Difference) specifies the 'difference' between two points in a circuit. EMF is the Potential Difference between the two terminals of the supply source.

Voltage is defined as the number of Joules per Coulomb. Or rather, the amount of energy contained by a unit of a charge. We measure Voltage in **Volts**. $V = \frac{W}{Q}$ and (more commonly) $V = IR$

Resistance (R) - The resistance of a component, or group of components, is defined as its opposition to the flow of current and thus a measure of its conductivity. We can determine the value of resistance for a component (measured in **Ohms**) by examining the PD (voltage) across the component and the flow of current. $R = \frac{V}{I}$

Power (P) - Power is defined as the rate of change of energy from one form to another. We therefore obtain this value from multiplying the Joules per Coulomb (V) by the Coloumbs per Second (I). This value is the most important for matters of power consumption. Power is measured in **Watts**. $P = VI$

Capacitance (C) - Also very important in creating power dissipation is Capacitance. Measured in **Farads**, and most often used in conjunction with the Capacitor, every conductive material has capacitance.

$C = \frac{Q}{V}$

Energy Stored in a capacitor, $W = \frac{1}{2}CV^2$

Time taken to charge capacitor, $T = \frac{Q_{total}}{I}$

2.2 CMOS, Transistors and Logic Gates

Underneath the collection of logic gates (the level of which most computer scientists understand circuits) we can see vast interconnections of transistors – the components from which logic gates (amongst other devices) are made. It is also the easiest place to observe how power is lost in a processor.

Modern microprocessors are implemented with Complementary Metal-Oxide-Semiconductor technology (CMOS). It is both a style of logic design and the set of industrial processes which are used in the implementation of these logic designs. The word complementary is used with reference to the use of pairs of transistors that 'complement' one another. Essentially, since both transistors are never conducting at the same time, when one transistor is active the output is connected to the supply voltage and when the other is active the output is connected to the ground (see Figure 2). CMOS currently allows the most dense transistor networks on a single chip.

CMOS chips use a combination of P-type (Positive) and N-type (Negative) MOSFETs [1].

In Figure 1 we can see the comparison between the function of the p-type and n-type MOSFETs. The binary digits represents input and output voltages of low or high. V_{dd} represents a supply voltage and V_{ss} represents a common ground. The switch diagram next to each transistor type shows the relation between its input and output. From this we can see that an n-type is in saturation (conducts between V_{dd} and V_{ss}) when its input voltage, V_{in}, is above the transistors switching threshold. A p-type, conversely, is in saturation when its V_{in} is below the switching threshold. The concept of the switching threshold is important for device timings since in high performance processors the time a transistor takes to reach this threshold is significant.

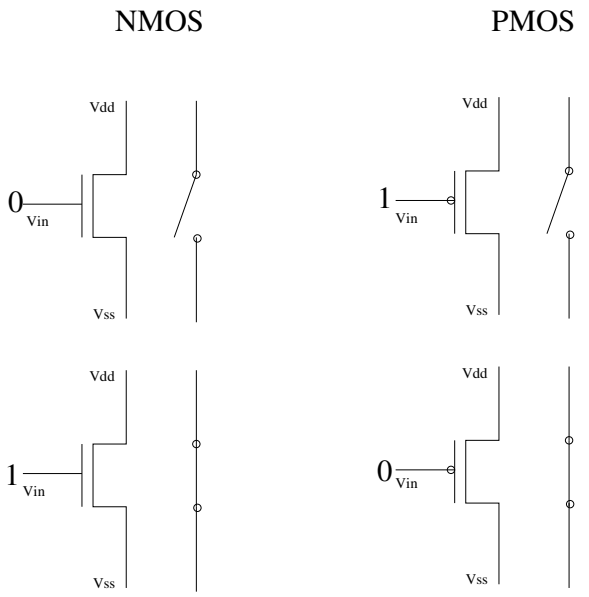

Figure 1: The two principle MOSFETs used in CMOS chips.

Figure 2 shows how we can arrange n-type and p-type transistors to create a logic gate – in this case a NOT Gate (inverter).

Now for some more indepth information:

Capacitance at V_{in} (input) will be the gates of the n-type and p-type transistors and the metal interconnect.

Capacitance at V_{out} (output) is created by the fanout (number of connections) to other gates and metal interconnects.

Propagation Delay is the time that the gate takes to charge the output to the correct switching threshold. This is key in configuring device timings.

[1] Metal Oxide Semiconductor Field Effect Transistor

NOT Gate

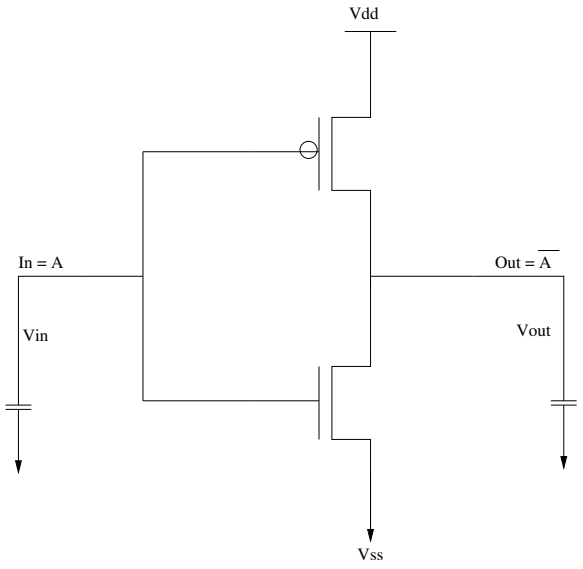

Figure 2: A NOT Gate constructed from p-type and n-type MOSFETs.

Switching Energy is defined as $W = QV$.

Load Capacitance Increases Propagation Delay – A higher fanout and interconnection will increase the amount of time taken to charge the output to the correct threshold.

For reference Figures 3 and 4 show two other configurations of CMOS logic, used here to create memory latches.

Latch

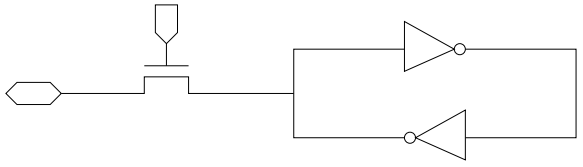

Figure 3: A latch constructed with two inverters. Holds value as long as power is supplied and actively drives output. However is fairly big since it requires 5 transistors. Commonly used for SRAM cells.

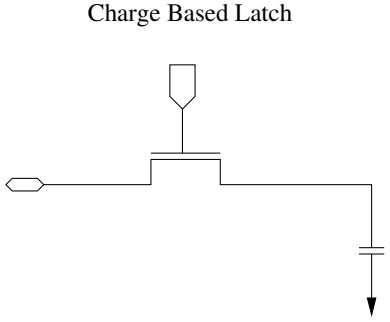

Figure 4: A charge based latch here constructed using one transistor and capacitor. Very small but charge 'leaks' off capacitor and reads can hence be destructive. Commonly used for DRAM cells.

3 Power Basics

Now that the low level details have been covered we can start to draw more general abstract information about power consumption in a processor. In a broad sense there are two types of power dissipation sources in a processor:

1. Dynamic Power –also called switching power.
2. Static Power – also called leakage power. Caused by transistor inefficiencies, this will result in a steady per cycle energy cost.

Dynamic power dissipation is the leading cause of power loss in a processor, however static power loss is becoming increasingly important given the increasing amount of transistors on an integrated circuit.

3.1 Dynamic Power Dissipation

Within dynamic power dissipation there are two causes for energy loss:

Capacitive Power Loss – Caused by charging/discharging of transistor output at transition from $0 \rightarrow 1$ and $1 \rightarrow 0$.

Short Circuit Power Loss – Caused by brief short-circuit current during transitions. This problem is due to the finite slope of input signals to transistors, the scope of which is beyond this document.

We can formalise dynamic power loss with equation 1 (the meaning of each term is show below) [2].

$$Power \sim \frac{1}{2} C V^2 f A \tag{1}$$

$C \rightarrow$ Capacitance. Function of wire length and transistor size. Measured in Farads.

$V \rightarrow$ Supply voltage. Measured in Volts.

$f \rightarrow$ Clock Frequency. Measured in Hertz.

$A \rightarrow$ Activity factor. How often, on average, do wires switch?

3.2 Static Power Dissipation

Static power dissipation due to leakage currents has two main dependency factors:

1. Gate oxide thickness.
2. Stacking and Input Pattern.

Since lowering leakage currents is a job for electrical engineers it will not be discussed further in this report, however one thing that static power dissipation does make clear is that making a smaller design does not only use less silicon space but it will also result in lower power dissipation.

4 Simple Power Reduction Methods

This section discusses some simple methods used by manufacturers and designers to lower the two main types of power dissipation.

4.1 Lowering Dynamic Power Dissipation

Given equation 1 for dynamic power dissipation the many general techniques used to reduce it include [9]:

Lower V_{dd} (V) – Given the V^2 element of the dynamic power consumption equation we can see that just lowering the supply voltage will have a quadratic effect. This has been a tactic in the design of integrated circuits by many chip manufacturers. However lowering the supply voltage also has a roughly linear degrading effect on performance.

Lower Capacitance at Gate Output (C) – Given the energy loss due to the output capacitance (a function of the fanout of other gates) it can be said that lowering this value will help decrease capacitance power dissipation, C.

Lower Clock Frequency (f) – This will have a linear effect on power dissipation however it will also affect the performance of the processor since the frequency dictates the number of cycles per second.

Reduce Activity Factor (A) – This is the key area in which processor designers can reduce the power consumption of their architecture designs. This is a function of the signal transitions against the clock rate. Clock Gating idle units and reducing how much units are used will help reduce this.

4.2 Lowering Static Power Dissipation

Many of the resulting solutions here are only useful for electrical engineers since they are extremely low level – from a design perspective limiting the size of architectures is the best we can do to avoid this form of dissipation, along with completely powering down functional units.

Use Fewer, Smaller Transistors – Stack transistors where possible to minimise contacts with V_{dd}.

Reverse Body Bias (dynamically adjust switching threshold) – Provides a low leakage sleep mode but maintains state (XScale).

V_{dd} **Gating** – Cut the supply or ground to a circuit. This will provide a zero dissipation sleep mode but all of the stored data will be lost. There are also overheads to consider when switching a unit on or off.

5 Effective Metrics

Something fundamental to analysing power dissipation in a system and assessing how effective a given modification is are metrics. This section briefly discusses the metrics used and the various pitfalls which one must be weary of, starting with some more simple metric definitions and finishing with a contemporary voltage independent metric. In each case the power is the power used to perform a given process and the time is the period of time this process takes to complete.

5.1 Power Consumption

- Determines battery life.
- Sets packaging size.
- $P = VI$
- Measured in **Watts**.

5.2 Energy Efficiency

- The rate of energy consumption.
- $Energy = Power \times Delay$ $(E = PT)$.

- Measured in **Joules**.
- A low energy value can be used to compare energy per operation at the same clock frequency (lower E is better).

5.3 Voltage Independent Metric

Power Delay Product – Specifies the average amount of energy consumed per switching event. $PDP = P_{average} \times t$

Energy Delay Product – This will take into account that increased delay can be conpensated for by lower energy transfer per operation. $EDP = PDP \times t$

However, given the quadratic effect of voltage on power efficiency and its metrics, we really need a way of measuring pure efficiency irrespective of voltage. This enables one to design a modification that should be as effective when implemented in any design, no matter what voltage the processor is running at.

The proposed metric [6], from a research group at the California Institute of Technology, and explanation is as follows:

$$Bu \rightarrow z \uparrow \tag{2}$$

$$Bd \rightarrow z \downarrow \tag{3}$$

The above production rules define the transition of a logic gate or operation z from one to zero. Given the above transitions let $E_{z\uparrow}$ and $E_{z\downarrow}$ represent the energy spent during each transition, respectively. This energy is dissipated during the charging and discharging of the capacitor associated with z (remember the associated capacitance of z can just be the fanout and interconnect to other gates). The associated energy used to charge a capacitor to voltage V is $C_z V^2$. As discussed earlier the energy stored in a capacitor is $\frac{1}{2}CV^2$. This means that the other half is dissipated as heat in the interconnect. Therefor energy loss during charging is:

$$E_{z\uparrow} = \frac{C_z V^2}{2} \tag{4}$$

Given V as the supply voltage. When the capacitor is discharged this energy is lost in the interconnect such that $E_{z\downarrow} = E_{z\uparrow}$.

The time $t_{z\uparrow}$ taken to charge the capacitor is the ratio of the final charge on the capacitor and the rate of charge transfer (current):

$$t_{z\uparrow} = \frac{Q_z}{i_z} \tag{5}$$

Where $Q_z = C_z V$. It is slightly harder to define the value of i since transistor current is difficult to analyse. However it can be made easier if it is assumed that the transistor in question is operating above threshold (not during velocity

saturation) then the current will either be linear or the saturation current. These are well defined and shown below:

$$I_l = k(2(V_{gs} - V_t)V_{ds} - V_{ds}^2) \tag{6}$$

$$I_s = k(V_{gs} - V_t)^2 \tag{7}$$

Now, if we assume that the voltages above vary proportionally with the supply voltage V then both the saturation and linear current depend quadratically on V. Thus current for both $i_{z\downarrow}$ and $i_{z\uparrow}$ is of the form $K_{z\downarrow\uparrow}V^2$.

Delay can now be formalised as

$$t_{z\uparrow} = \frac{C_z}{K_{z\uparrow}V} \tag{8}$$

If the above equation for delay is combined with the equation for energy loss it can be seen that the term Et^2 is independent of any voltage term [6].

Since this is just a further extension of the Energy Delay Product, it is often referred to as the ED^2P metric, EDDP.

$$ED^2P = Et^2 \tag{9}$$

With this the processor designer can balance both energy and time to perform a process, with increased time being compensated for by decreased energy readings and so on. When comparing two designs a lower value for EDDP is the most desirable when considering power efficiency.

It is also worth noting that $\frac{MIPS^3}{P}$ is inversely proportional to the Et^2 metric. For an explanation of this see Appendix 1, where it is also explained how we can thus use $P \times CPI_{Average}^3$ as a voltage independent metric for comparing energy efficiency.

6 Analysis and Tools

When designing processors/circuits for power efficiency there are various tools that can be used [10], each one implemented to monitor power at a different level of abstraction. Discussed briefly below are some of the more popular (and effective) of these tools [1].

6.1 Circuit Level Models (HSpice)

Modelling at this level involves creating simulations that are as similar as possible to the actual implementation of a circuit in silicon.

One of the most popular (and famous) tools used for this purpose is HSpice. HSpice models diffusion, gate and wiring capacitance. These analogue simulations are performed by:

- Large detailed device models created from empirical measurement.
- Solving large systems of analogue electrical equations.

The result of this complexity is an often accurate (power dissipation can be estimated to within a few percent) but extremely slow simulation. This makes HSpice ideal for testing implementations of 10-100 thousand transistors, but impractical for anything larger than this.

The tool Powermill is similar but slightly less accurate. However the reduced accuracy comes with the benefit of around a 10x increase in speed.

6.2 Logic Level Models

Logic level models obtain switching information for each signal and logic transition in a circuit. Modelling behaviour at this level is generally achieved using hardware description languages such as VHDL and Verilog with their associated tools.

From these behaviour models a simulator can generate capacitance estimates by:

- Estimating the gate size of the specified behaviour model, similar to actual synthesis.
- Creating wire load estimates based on this.

The resulting switching and capacitance information provides dynamic power estimates only. These results will generally be less accurate than circuit level models but much more scalable to very large designs.

6.3 Architecture Level Models

The most commonly used level for computer architecture design has two approaches in its simulation:

1. Event Based Modelling – Interface some high level power models with cycle based simulations.
2. Instruction Based – Power estimates created based on instruction usage.

Both of these types of simulation can be implemented using empirical circuit design measurements as estimates or general capacitance models.

Since Power $\sim \frac{1}{2}CV^2fA$ these two approaches estimate CV^2f in different ways:

Capacitance Models – Estimate CV^2f using analytical models. This requires estimating 'wire' length and the size of transistors.

The popular tools used to do this include: Wattch [3] [4], PowerAnalyzer and Tempest.

Empirical Models – Estimate CV^2f using measurements taken from actual implementations and adapt them to model the current circuit.

Often internal corporate tools are used to do this, however the two main publicly available tools here are PowerTimer and AccuPower.

For large scale designs the general accuracy of architecture level simulation is sufficient and indeed is the only realistic option for general high level comparison of designs.

7 Summary And Current Issues

The current approach to increasing the energy efficiency of processor designs, in a general sense, is an amalgam of the previous sections. Initially the engineer needs to understand the causes of power dissipation and then make some intuitive early stage decisions and create different designs. With these different designs the engineer then needs to run simulations to decide where power is being dissipated most in his approach(es) and apply a suitable metric to them – the most common and useful of which is generally an Et^2 metric. Once this dissipation has been quantified it is possible to try out different implementations that may reduce this power dissipation.

The most potent method of reducing power consumption at the architecture level is to focus on reducing the activity factor of different units in the processor, by means of intelligent alogorithms. This is largely due to the extensive research carried out into reducing gate level power dissipation.

Another area worth considering outside of processor design, which can have great effect, is power aware operating system algorithms. Optimisations at the application level will often result in the greatest power efficiency. They are however highly dependent on user input and application specific. As a result intelligent algorithms at the operating system level have the possibility to greatly decrease power dissipation by reducing the activity factor to a minimum. Research in this is still youthful.

8

References

[1] D. Brooks. Power Aware Computing Notes. Technical report, Harvard University, USA, sep 2004.

[2] D. Brooks, P. Bose, S. Schuster, H. Jacobson, and P. Kudya. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. IEEE Micro, November 2000.

[3] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. 27th annual international symposium on Computer architecture, 2000.

[4] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, University of Wisconsin-Madison, 1997.

[5] Canturk Isci and Margaret Martonosi. Run-time Power Monitoring and Estimation in High-Performance Processors: Methodology and Experiences. Micro-36, dec 2003.

[6] Alain J. Martin, Mika Nystrom, and Paul L. Penzes. ET^2: A Metric for Time and Energy Efficiency of Computation. 2003.

[7] Dharmesh Parikh, Kevin Skadron, Yan Zhang, Marco Barcella, and Mircea R. Stan. Power Issues Related to Branch Prediction. IEEE HPCA, 2002.

[8] Myke Predko. *Digital Electronics Demystified*. McGraw-Hill, New York, 2005.

[9] John S. Seng and Dean M. Tullsen. Exploring the Potential of Architecture-Level Power Optimizations. PACS, 2003.

[10] Viji Srinivasan Michael Gschwind Pradip Bose Philip N Strenski Victor Zyuban, David Brooks and Philip G Emma. Integrated Analysis of Power and Performance of Pipelined Microprocessors. *IEEE Transactions on Computers*, 53(8), aug 2004.

[11] V. Zyuban and P. Kogge. Optimization of High-Performance Superscalar Architectures for Energy Efficiency. ACM, 2000.

[12] Victor V. Zyuban. *Inherently Lower-Power High-Performance SuperScalar Architectures*. PhD thesis, University of Nore Dame, Indiana, mar 2000.

9 Appendix 1

Relation of $\frac{MIPS^3}{P}$ to Et^2.

$$\frac{MIPS^3}{P} \propto Et^2 \tag{10}$$

$$Et^2 = (P \times t) \times t^2 = Pt^3 \tag{11}$$

Where $P =$ Power in Watts, MIPS = Millions of Instructions Per Second, $t =$ time in seconds for the process to complete and $E =$ energy used for the process.

$$t = Instruction_{Count} \times CPI_{Average} \times t_{percycle} \tag{12}$$

$$MIPS = \frac{Instruction_{Count}}{t} = \frac{Instruction_{Count}}{Instruction_{Count} \times CPI_{Average} \times t_c} = \frac{1}{CPI_Average \times t_c} \tag{13}$$

Since t_c is fixed by the frequency of the processor it does not need to be considered at the architecture level of designing energy efficient processes and will be a constant in these equations.

The instruction count is inversely proportional to the CPI so also can be removed from the equation for Et^2.

This leaves us with the following more obvious relationship:

$$\frac{(1/CPI_{Average})^3}{P} \propto P \times CPI^3_{Average} \tag{14}$$

This shows that what we are really measuring is the average power usage of a process in relation to its CPI cubed. The following statement should appear more intuitive (perhaps) to the reader:

$$EnergyEfficiency \sim P \times CPI^3 \tag{15}$$

HTracer V0.5: A User Guide

M.A. Hicks, C. Egan

July 2005

1 Introduction

This document details the usage of the Hatfield Tracer (HTracer), from a user's perspective. HTracer is a cross platform dynamic instruction stream research tool that allows one to produce detailed traces of the dynamic instruction stream of almost any compiled binary program and requires no special compilation procedure, with respect to the programs that are to be traced. After a trace is complete, relevent execution statistics are also supplied.

In essence, one simply supplies HTracer with the binary program that they which to be traced. Using a number of switches and flexible masks, almost any required information can be retrieved pertaining to the behaviour of that program at runtime.

This document is structured as both an explanation of what HTracer can do, and then an explanation of how this can be achieved. Before this, however, some basic concepts and requirements are detailed so that anyone unfamiliar with a tracer can understand this document's content. Some example output is also supplied to show what can be expected from a trace when run with certain parameters.

It is recommended that this document be read from start to finish; although the document is divided into sections, they are not completely discrete and each contains information useful to using HTracer.

2 What Is A Tracer?

Generally, most computer scientists write their software in a high level language, such as C/C++ or Pascal. At this level it is easy to understand and imagine the thread and behaviour of the program were it to be executed by some theoretical machine. However, in reality, this high level language will be compiled to a much lower level format. This format is known as machine code, since it is the binary format of 'simple' atomic instructions executed by the CPU of a computer.

At this low level the behaviour becomes much more complex than one had imagined at the higher level, with seemingly new patterns and behaviour appearing in the dynamic execution of code. It is with this in mind that HTracer was created;

to allow the monitoring of the dynamic instruction stream in a useful and flexible manner.

Hence it can be said that a tracer, and in particular HTracer, will intercept the real time execution of machine code and produce data records detailing the sequence of program execution.

3 Requirements

Due to its low level nature, HTracer is implemented at the kernel level of the operating system. As such HTracer will function on almost any Unix-like operating system, but in particular has been designed with Linux in mind. Given the availability of Linux on a number of different architectures, HTracer can be used to produce execution traces on many processor architectures including, but not limited to: x86, AMD64, PowerPC and Sparc. This cross platform ability is one of the key advantages and interest points of HTracer, since it allows the comparison of static to dynamic code behaviour across different architectures.

In the current version of HTracer (0.5), the recommended system requirements are as follows:

- A machine with a working installation of Linux.
- Kernel version $>= 2.6$.
- Since tracing is by its very nature a slow process, a fast CPU is always handy!
- A working understand of Linux and the command line is assumed.
- A copy of the tarball archive "htracer.tar.bz" .

Assuming one has met these requirements, the next section details how HTracer can be installed on a machine.

4 Installation

Firstly the tarball archive needs to be decompressed. At a command prompt, change to the directory of the tarball and run the following command:

```
$ tar xfj htracer.tar.bz
```

This will decompress all of the source code into a directory called HTracer. Change into that directory before proceeding. There will be a file present here called "README", which contains release notes and technical details for the current release.

The next step is to compile HTracer to run on the current machine. This can be achieved with the following commands:

```
$ make
```

This starts the build process. If this fails, it is likely that some configuration options need to be changed from their default values. Execute the following command to open the configuation file:

```
$ make configure
```

Read through the comments next to each option and make the appropriate change for the current architecture. This process is detailed in the "README" file. When this has been completed, execute the first compilation step to compile.

To install HTracer globally (recommended) execute the following make command:

```
$ make install
```

The command "htracer" should now work at the command prompt on the current machine. HTracer has been successfully installed.

5 Usage

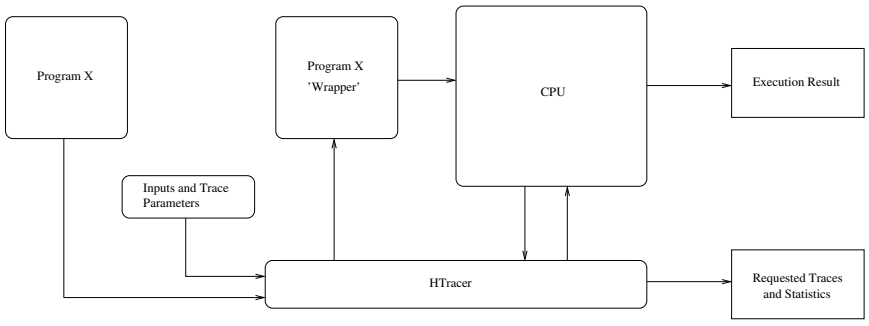

Figure 1: Logical block diagram of the HTracer tool.

HTracer permits the masking of almost any information that is required during program execution. That is to say, HTracer can log various state information for, perhaps, only branch instructions or arithmatic instructions. This is particularly useful since full traces of programs are often not required (and take considerably more time and disk space).

To achieve this functionality, HTracer employs a system of masks and execution parameters. These are all invoked in one command, at the point of starting the trace, and are explained in the proceeding two subsections.

The overall structure of the command to execute a trace is:

```
$ htracer [-fvsirome] <program> <program-parameters>\
 <output.file> <instruction.mask>
```

5.1 Parameters

Here is an explanation of what each parameter in a trace execution means:

-fvsirome – These are the switch parameters to the tracer which specify how is should behave and what sort of information should be logged. These switches are explained shortly. Any number of the switches can be used.

program – The relative or absolute location of the executable to be traced. e.g. /bin/ls

program-parameters – This should be a space seperated list of parameters that one wishes to run the traced program with. These MUST be encapsulated by speech marks. e.g. "-lh /home/mike"

output.file – The file where all of the tracers output should be dumped to. This option is not used when the "-o" option is specfied. Output is sent here if the instruction at the current PC matches a mask OR if one specified the full trace option.

instruction.mask – The name of the file containing the instruction masks and groups (if the option "-m" was given to the tracer). See the Masks subsection for further information on masking.

The following list describes the meaning and function of each of the switch parameters mentioned in the previous list:

-f Run a full trace. Information for every instruction will be dumped into the output file. If the use of a mask is specified, it will still be used, however the tracer will not ignore instruction that are not specified in the mask.

-v Verbose. Provides some extra information about the progress of the tracing. On some architectures this includes information about the number of clock cycles that the tracer ran for.

-s Outputs some statistics at the end of the trace. The statistics include information about the occurence of each instruction group specified in the mask.

-i Log the actual instruction fetched during each step of the trace. This is useful if one needs to examine any matched instructions and perform any post trace analysis.

-r Log ALL of the user registers at each matched instruction. This option is currently (V0.5) platform dependent and will not function correctly on all architectures. Using this option also significantly slows down the execution of a program trace.

-o Instead of using an output file, send all of the dumped information to stdout. Useful if it is needed to redirect the output to somewhere else, perhaps directly into another program.

-m Specifies the use of a mask file. See the next subsection.

-e A line is written at the top of the tracer output which explains the format of the tracer log.

In the example output section, the relevent command used to produce the example output is shown to illustrate how trace options affect trace output.

5.2 Masks

In order to monitor a program, and in order to trace it accurately, HTracer single steps each instruction through the entire processor pipeline. This means that every instruction is moved through the whole pipeline, one at a time, and then the state of the processor is checked. This must be done since it would be difficult to trace a program when multiple instructions were being executed at any given time. As a direct result of this, program traces produced by HTracer take significantly longer to run than simple executing the program natively.

The penalty of single stepping instructions cannot be reduced. However another key contributor to trace production time is the amount of information being written to disk in each step. For most full trace producing applications, this means a full trace and data being dumped every cycle. This is slow. To try and ameliorate this problem and to produce more meaninful labelled output, selective instruction masks have been implemented. These masks match certain instruction appearance formats which are completely specified by the user.

All instruction masks are grouped into user specified groups; these groups are really just a way of collecting together similar instructions, to be reported under the same name (if this is desired). For instance it may be useful to have all load instructions reported under one term (given the variety of these present on the x86 architecture). As well as appearing under these groups in the trace output, it is also how program statistics at trace completion will be shown.

Figure 1 shows how HTracer spawns a 'wrapper' child process and executes this on the CPU, whilst monitoring it at each step. It is at each step that the instruction masks are compared with the instruction currently being executed. If a match occurs, the requested trace data (see parameters subsection) is stored in the specified file. If no match occurs, then the instruction and processor state is ignored. This will become more apparent after reading the rest of this document.

The format of a masking file is as follows:

```
# Any line beginning with the hash symbol is ignored.
% <group name> <1|0: dump next pc>
> <hex bit appearance> <hex bit mask>
```

```
> ..... ........
> ..... ........

%<another group name> <1|0: dump next pc>
> <hex bit appearance> <hex bit mask>
> ..... ........
> ..... ........
```

A group consists of one of more instruction mask pairs. The name of the group is completely arbitrary but a good choice is a name that reflects the common theme of all of the instructions contained in the group. For instance "branch" for a group containing branch instructions. There can be any number of instruction groups. A 1 or 0 after the group name specifies whether the tracer should log the next value of the Program Counter after a match with one of the masks occurs. This is useful for logging branch targets.

The function of the mask pairs are:

Hex Bit Appearance – This specifies how certain parts of the instruction should appear. In essence, the actual instruction appearance is written here, for the bits that are to be monitored (see next item).

Hex Bit Mask – This hex sequence specifies which parts of the previous hex appearance should actually be compared with the currently executing instruction (similar to a subnet mask, for instance).

An example mask, to catch x86 branch instructions, would look something like this:

```
# x86 Instruction format tracer mask.
# Currently matches only branch instructions.
# Remember little endian byte ordering.

% Branch 1
> 00000070 000000f0
> 000000e3 000000ff
> 000000eb 000000ff
> 000000ea 000000ff
> 000000e9 000000ff
> 000000ff 000000ff
> 0000000f 000000ff
```

Taking the first mask pair here as an example, only instructions where bits 4-8 equal 7 would cause a match and a consequential state dump to the output file. This is because 00000070 says that bits 4-8 should equal 7. The rest of the bits here show as zero; this is because they are ignored by the actual mask applied which

means the tracer only examines bits 4-8 for this mask. Given this explanation it is clear that the other mask pairs will match. All of these masks will report instructions under the term "Branch". If they needed to be seperated in the tracer output then they could simplt be specified under seperate groups in the mask file (one mask pair per group). The "1" after the group name "Branches" means that the effective target of each of the logged branch instructions will also be stored. This is demonstrated in the next section.

6 Tracer Output

During the execution of a program trace, HTracer will store the state of the processor, as requested, to either the specified output file or the standard output. Regardless of which option is chosen, the output will always appear in the same format, as described here.

It is worth remembering, from the parameters section, that a line can be written to the start of the tracer output which specifies the layout of the columns in the output file (so that one doesn't need to remember).

The tracer output file is arranged into columns and rows. Each row specifies one state of the CPU and thus one instruction execution. Each column specifies a value of something within that state. Here is a description of the format of the columns in the tracer output:

```
<PC> <GROUP> { (<INSTRUCTION>)? (<REGISTERS>)? (<NEXTPC>)? }
```

PC The program counter value of the instruction being executed in hexadecimal. By default it is the relative program address i.e. 0 is the base address in memory of the program being traced.

GROUP The name of the instruction group for which this instruction matched. N.B. if a full trace is to be run, then all instructions will appear here, regardless of whether or not they match a group mask. In this case, instructions which did not match a mask will appear in the "Generic Instruction" group.

INSTRUCTION If requested at trace execution, this is the actual instruction being executed in hexadecimal. This is extremely useful for performing post trace analysis and further decoding any traced instructions.

REGISTERS If requested at trace execution, the whole general purpose register file. This is output as a comma delimited hexadecimal list of the value contained in each of the general purpose registers. These values are useful for more complex post trace analysis.

NEXTPC If specified in the mask file for a particular instruction group (see masking section), then the next PC value will be stored here.

Given the previous specification, here is a snippet of tracer output, created using the example mask in the previous section and the parameters **"-sime"**. The full command executed to produce this output:

```
htracer -sime ./jumper "100000" traceroutput.txt x86branchmask.txt
```

```
....
<PC> <MNEMONIC> <INSTRUCTION> <NEXTPC>
0006eabd Branch fe35840f 0006eac3
00064200 Branch 0038a3ff 000b65c0
000b65dc Branch 009e860f 000b65e2
000b65f0 Branch 4616b60f 000b65f3
000b65f7 Branch 4101b60f 000b65fa
000b6600 Branch 0fc0940f 000b6603
000b6603 Branch f655b60f 000b6607
000b660a Branch 09c2950f 000b660d
000b6611 Branch 009b850f 000b66b2
000b66b2 Branch f645b60f 000b66b6
000b66b6 Branch f755b60f 000b66ba
0006eadb Branch 958b1075 0006eaed
....
```

The first column shows the PC, the second is the instruction group, the third is the actual instruction and the final column is the value of the next PC.

After the trace has completed, if requested at execution, some statistics about the trace will be displayed. In the case of this example, the statistics were as follows:

```
-------------------------------------------
         Program Statistics
-------------------------------------------
./jumper
-------------------------------------------
Ins Class ----> Number of Instructions | %

Generic Class   789495 | 78.20%

Branch          220057 | 21.80%

Ins Total:      1009552
-------------------------------------------
```

The format of the statistics is largely dependent on the mask file used. One entry is shown per instruction group specified in the mask file, showing how many times any group was matched, both as a value and as a percentage of the total instructions.

It is, at this point, worth noting that the likely output file generated by HTracer for any realistic program or benchmark is very large. For instance in an intensive benchmark trace, an output file can be gigabytes in size. There must be enough free space on the target storage disk to accomodate the entire trace.

7 Known Issues

There are a number of known issues/problems with HTracer which have yet to be addressed.

1. Library initialisation code is traced. At the start of a program's execution, all of the libraries used by the program are initialised and mapped in memory. Currently (V0.5) HTracer will trace this code. This may not be desirable. A work around for this in the meantime is to create a simple blank program that uses all of these libraries, trace it using HTracer, and thus work out how many instructions at the beginning of the trace can be ignored. Alternatively (and easier), build the program to be traced so that it is statically linked, thus removing any need for library initialisation.

2. The logging of the entire register file is currently (V0.5) not fully implemented on all architectures.

3. Tracer output to a file cannot be split across several different files/disks. A work around for this is to send the trace output to the standard output and pipe it to one of many standard *nix programs which can reroute the output to several different files.

8 References

The following sources were used, to varying degrees, during the construction of HTracer.

References

[1] Intel. *Intel Architecture Software Developer's Manual*, September 2004.

[2] S.Sandeep. Process Tracing Using Ptrace. *Linux Gazette*, 81, 2002.

[3] Various. *Linux Operating System Manual Pages*, 2005.

[4] Various. The Linux Kernel Archives. Online, 2005.

[5] Various. Tux.org Discussion Lists. Online, 2005.

Appendix C: Additional Background

Appendix C: Additional Background

1 Electronics and Modern Transistors

Computer scientists are used to recognising abstract models of processor behaviour, but these are typically centralised around performance [1] [2]. In order to better understand power dissipation [3] it is important that the underlying principles are set out. The following subsections explain transistor, and hence gate level, power dissipation of modern nanometre circuits in sufficient detail to understand this work. However, a more indepth discussion is supplied in Appendix A "An Introduction to Power Consumption Issues in Processor Design" [4].

1.1 Basic Electricity Nomenclature

All electronic circuits function by manipulating the flow of electricity. As a result, various terms will be used regularly in this chapter and throughout this dissertation. Although these terms are elementary, it is important that they are included for clarity [5] [6]. Figure 1 shows a very simple circuit diagram illustrating the schematic voltage drop across a lamp in a circuit. V represents the voltage drop, and A represents the current flow. This aids in the explanation of the terms shown below.

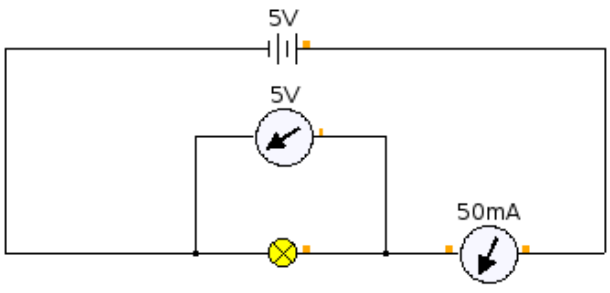

Figure 1: A very simple circuit

Electron – A negatively charged subatomic particle that is capable of carrying electrical energy.

Electricity – In the most general sense, a flow of electrical charge in a material. Specifically, in electronic circuits, it is the flow of electrons through a conductor from a negative to positive terminal. Different materials have varying degrees of free electrons which affect the rate of flow.

Electrical Energy (W) – An abstract measure of the potential energy stored or provided by a circuit. As with all energy, electrical energy is measured in Joules.

Charge (Q) – In an electronic circuit, charge is a measure of a number of charge carriers in a given area, or passing a particular point over a given time period. Measured in Coulombs. A charge of 1 Coulomb corresponds to 6.3×10^6 electrons.

Current (I) – The current at a given position in a circuit is a measure of the charge passing through that point per second. Measured in Amperes. 1 Ampere is equal to 1 Coulomb of Charge per Second.

Voltage (V) – The amount of Electrical Energy stored per unit charge. Measured in Volts. 1 Volt is equivalent to 1 Joule per Coulomb. Voltage is also used to quantify how much Electrical Energy is lost or gained across a particular component or section of circuit.

Resistance (R) – A measure of the opposition to the flow of current in a particular material or section of circuit. Measured in Ohms. The more conductive a material is, the lower the resistance.

Capacitance (C) – Conductive materials have the capacity to store electrical charge (and thus energy). Capacitance is a measure of this capacity for a component, and is measured in Farads.

Power (P) – A measure of the Electrical Energy transferred by a given current. Measured in Watts. Equal to Voltage multiplied by the Current flowing.

Power Source – A power source is capable of supplying a finite amount of electrical energy to a circuit. Typically, when examining processor energy efficiency, this power source will be a battery. The rate at which the power source is drained will set its lifetime.

Power Dissipation – This refers to the power consumed by a circuit in an inefficient manner, or rather, power consumed which is not supporting the logical behaviour of the circuit. A completely efficient circuit would dissipate no power at all.

1.2 Transistors, Logic Gates & CMOS

Central Processing Units (CPUs) are usually modelled at the architecture level by using structural diagrams and, if more detail is required, with the use of logic gates. Modelling circuits at these levels of abstraction makes their behaviour easier to understand but also detracts from the physical transistors which are supporting this behaviour. While an individual transistor may appear very efficient, collectively they result in significant power dissipation; a modern processor, such an AMD Dual Core Athlon 64 [7], will contain as many as 233.2 million transistors. The following subsections explain how this supporting structure of transistors result in power dissipation and other negative effects [5].

1.2.1 MOSFETs

Metal Oxide Semiconductor Field Effect Transistors (MOSEFETs) are the most commonly used type of transistors in modern VLSI circuits. MOSFETs are constructed in silicon using a combination of n-type and p-type material (discussed under doping). The effect of an electric field on these types of semiconductor material is capable of manipulating the flow of current, and their combination can thus behave like a switch.

Semiconductor A solid material with variable electrical conductivity properties that can be varied either permanently or dynamically. Silicon is the most prominent and widely used example of such a material.

Doping A process where impurities are intentionally introduced into a semiconductor material in order to change its electrical properties. Highly doped material is often shown using 'N+' or 'P+' – the + denoting that the material is highly doped. A 'normally' doped semiconductor, as described below, will have a majority charge carrier, but will still contain a minority of the opposite charge carriers. This is important to the behaviour of transistors.

n-type Semiconductor material that has been doped using a substance that increases the number of free negative-charge carriers. This substance is commonly referred to as 'donor material' as it 'donates' electrons to material into which it is introduced.

p-type Semiconductor material that has been doped using a substance that increases the number of free positive-charge carriers.

Figure 2 shows a cross section schematic of how n-type and p-type semiconductor material can be combined into an NPN transistor. The three letter initialism represents the doping structure of the transistor. The source and drain terminals are connected to two areas of highly doped n-type material (N+). The main body of the transistor consists of p-type material (P). The gate terminal is separated from the transistor by an oxide such that only the gate's electric field may affect the body.

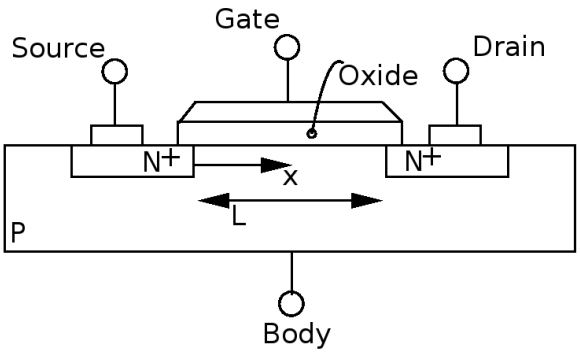

Figure 2: An NPN transistor where P shows p-type material, N+ shows highly doped n-type material

When a positive voltage is applied to the gate terminal, a process referred to a tunnelling takes place. The field effect of the positive voltage at the gate terminal causes the majority positive-charge carriers in the p-type base to be repelled while the negative-charge carriers are attracted to form a 'tunnel' between the two highly doped n-type junctions (L). The result is that there is now a negative-charge carrying area between the source and drain junctions; a current can flow between the drain and source. The voltage required to allow current to flow in the transistor is known as the 'Threshold', and is set by the designed chemical properties of the doped semiconductor.

A PNP transistor can be constructed by reversing the doping types shown in Figure 2. This type of transistor will function, when the gate terminal is connected to a negative terminal voltage, by creating a tunnel of positive charge carriers which allow a current to flow between the drain and source.

1.2.2 CMOS

Complimentary Metal-Oxide-Semiconductor (CMOS) is a technology process used to create most modern processors from MOSEFETs. In CMOS, logic gates are constructed using a 'complementary' arrangement of NPN and PNP MOSFETs (see Figure 3. In a typical CMOS gate, the NPN MOSFET part will control the connection of the output to the ground (V_{dd}) of the circuit; conversely, the PNP MOSFET part will control the connection of the output to the high voltage (V_{ss}).

Figure 4 shows how a CMOS Not-Gate is constructed from NPN and PNP transistors. Its complementary structure using two transistors ensures that the output is always swiftly drained or charged to the correct level by the effect of both transistors always driving their output to opposite levels (when connected to the same input); when the NPN is 'on' the PNP is 'off'. This means that, for a given CMOS circuit, there is no set of inputs that will cause a continuous current from source to drain and also results in faster switching times.

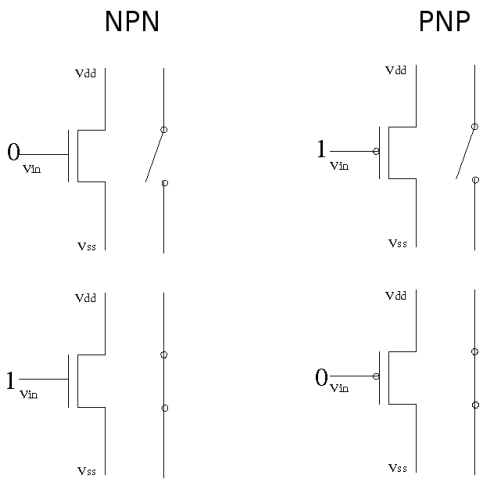

Figure 3: The CMOS logical representation of NPN and PNP transistors with the associated switch behaviour

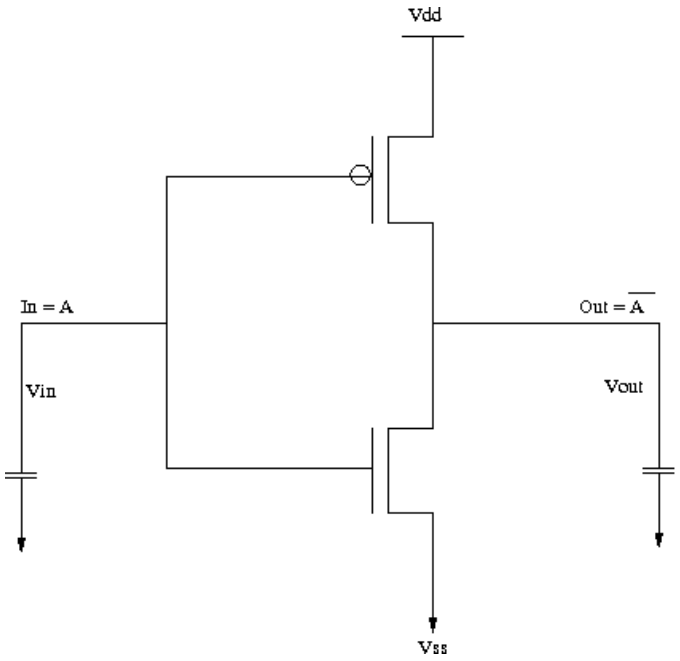

Figure 4: An example CMOS Not-Gate

References

[1] Egan, C.: Dynamic Branch Prediction In High Performance Super Scalar Processors. PhD thesis, University of Hertfordshire (August 2000)

[2] Parikh, D., Skadron, K., Zhang, Y., Barcella, M., Stan, M.R.: Power issues related to branch prediction. In: IEE High Performance Computer Architecture. (Februrary 2002) 233–244

[3] Gonzalez, R., Horowitz, M.: Energy dissipation in general purpose microprocessors. IEE Journal on Solid State Circuits **31**(9) (September 1996)

[4] Hicks, M., Egan, C., Quick, P., Christianson, B.: An introduction to power consumption issues in processor design. Technical report, University of Hertfordshire (July 2005)

[5] Amos, S., James, M.: Principles of Transistor Circuits. Butterworth-Heinemann (1999)

[6] Horowitz, P., Paul, Hill: The Art of Electronics. Cambridge University Press (1989)

[7] AMD: Athlon 64 X2 Processor Manual. (2006)

Appendix D: Raw Data

Raw Results Data

The results on the following pages are the raw data for the major experiments conducted and presented in this dissertation. They are presented here in order to reinforce the veracity of the average results shown at the end of each experimental section or chapter. There are five main result sets:

Raw Scalar Data – The raw data collected for the experiments conducted against the scalar processor baseline

Raw 2Way Issue Data – The raw data collected for the experiments conducted against the two-way issue processor baseline

Raw 16Way Issue Data – The raw data collected for the experiments conducted against the sixteen-way issue processor baseline

Raw Fixed Bias Data – The raw data collected for the experiment conducted to examine the use of a fixed bias hinting level

Raw BTB Resize Data – The raw data collected for the experiment conducted to examine the potential savings achievable by resizing the BTB

Each of the columns in the five results sections presents one of the following values (by column title):

Benchmark – The name of the EEMBC benchmark

SBTOTAL – The total number of static branch instructions in the benchmark

SBHINTED – The number of static branches for which a hint was applied by the combined algorithm

DINSTOT – The total number of dynamic instructions executed

DINSNEW – The total number of dynamic instructions executed after the application of the combined algorithm

DBPOLD – The number of accesses made to the dynamic branch predictor during the benchmark's execution

DBPNEW – The number of accesses made to the dynamic branch predictor during the benchmark's execution after the application of the combined algorithm

PWROLD – The total average power used by the entire processor per executed instruction

PWRIDEAL – The total average power used by the entire processor per executed instruction if the branch predictor consumes no power

PWRNEW – The total average power used by the entire processor per executed instruction after the application of the combined algorithm

INSCOMMIT – The total number of instructions committed during the execution of the benchmark

INSCOMMITN – The total number of instructions committed during the execution of the benchmark after the application of the combined algorithm

B_PWR_OLD – The average power per executed instruction consumed by the dynamic branch predictor

B_PWR_IDE – The average power per executed instruction consumed by the dynamic predictor under ideal circumstances (i.e. zero)

B_PWR_NEW – The average power per executed instruction consumed by the dynamic branch predictor after the application of the combined algorithm

CYCLES – The number of cycles for which the benchmark executed

CYCLESN – The number of cycles for the benchmark executed after the application of the combined algorithm

Raw Scalar Data

Benchmark	SBTOTAL	SBHINTED	DINSTOT	DINSNEW	DBPOLD	DBPNEW
a2time01	222	60	41549	42139	8081	7319
aifftr01	203	57	4914484	4889046	618784	239710
aifirf01	196	52	69899	68434	11038	8483
aiifft01	193	51	4716593	4942761	594585	251154
basefp01	170	39	57676	57653	10412	7844
bitmnp01	353	70	346365	346744	55428	35771
cacheb01	187	34	224145	222942	31876	13942
canrdr01	237	46	997677	985405	227395	69888
idctrn01	277	45	402930	401975	48438	30168
iirflt01	240	89	61575	61553	10141	8581
matrix01	303	54	2560154	2341943	464366	211186
pntrch01	201	54	231860	229998	52427	19257
puwmod01	255	36	1590409	1570416	365212	108887
rspeed01	168	37	357576	343202	81178	26273
tblook01	186	56	102900	103557	22054	17637
ttsprk01	305	36	677040	669566	153507	49653
cjpeg	1605	495	64372423	64373440	6693144	951694
djpeg	1871	605	88392767	88390600	12757158	6450502
rgbcmy	157	39	107249021	107722658	23831899	7226389
rgbhpg	135	27	24060459	23928550	3074971	775380
rgbyiq	154	36	56309529	56308682	8762870	5997811
ip_pktcheck	1032	64	11540506	11581303	2601518	773116
ip_reassembly	1159	143	14067731	14105818	2838471	969862
nat	2197	282	13893181	13935740	3232061	1033761
ospfv2	1018	64	2889122	2895624	823721	137424
qos	1957	248	24732446	24720064	5173931	3347996
routelookup	1011	61	5075891	5022486	1204430	366426
tcp	1402	171	135502	134465	19368	17325
bezier01	145	26	4395358	4375263	481488	204679
dither01	151	28	10802113	10802102	1741737	1237889
rotate01	219	97	4101269	4089950	1053705	354865
text01	340	113	6176652	6152184	1501590	801101
autcor00	143	29	802917	802114	73933	25367
conven00	144	27	484479	484467	78542	58550
fbital00	155	32	964054	964242	145226	72338
fft00	170	39	310996	311422	37637	20116
viterb00	159	29	954995	954934	90636	45818

Raw Scalar Data

PWROLD	PWRIDEAL	PWRNEW	INSCOMMIT	INSCOMMITN	B_PRW_OLD
15.44	14.3	15.25	41111	41743	0.22
10.72	10.04	10.41	4814678	4814680	0.26
14.06	13.13	13.78	69310	67900	0.21
10.68	10.02	10.31	4618154	4834126	0.26
14.72	13.59	14.29	57200	57200	0.23
11.31	10.39	11.01	340230	340276	0.3
12.14	11.44	11.79	219942	219942	0.23
10.15	8.61	9.26	958477	958477	0.53
11.17	10.36	10.78	401673	400825	0.27
14.57	13.58	14.32	60947	61014	0.2
10.15	8.85	9.48	2554423	2337796	0.48
11.66	10.25	10.79	226803	226363	0.43
10.05	8.61	9.15	1526642	1526642	0.54
10.6	9.16	9.48	344117	333981	0.49
12.3	10.93	11.99	99968	100598	0.36
10.23	8.83	9.42	649657	650289	0.51
10.92	10.21	10.27	64220166	64218532	0.22
10.19	9.42	9.71	87016324	87015471	0.33
9.9	8.56	8.99	103180826	103178575	0.54
9.59	9	9.19	23619293	23620126	0.31
9.33	8.59	9.06	54873160	54872316	0.39
10.21	8.8	9.22	11149713	11150956	0.53
10.72	9.63	10.09	13696229	13697048	0.43
10.43	9.04	9.59	13465421	13464651	0.5
10.95	9.22	9.47	2821506	2821541	0.59
11.9	10.57	11.44	24689707	24689762	0.41
10.18	8.69	9.25	4865802	4865526	0.56
13.84	12.95	13.75	134703	133746	0.21
10.64	10.11	10.42	4330053	4329246	0.2
10.02	8.99	9.69	10629212	10629166	0.4
11.45	10.06	10.37	4005387	4006235	0.42
11.28	9.79	10.68	6056185	6055422	0.48
10.32	9.7	9.83	800604	799751	0.22
10.17	9.32	9.96	478189	478180	0.34
9.99	8.93	9.44	961588	962151	0.41
10.72	10.03	10.43	307089	307144	0.24
10.07	9.47	9.71	950867	950867	0.21

Raw Scalar Data

B_PWR_IDE	B_PWR_NEW	CYCLES	CYCLESN
0	0.19	141831	144075
0	0.11	9617038	9600434
0	0.16	201838	198265
0	0.11	9274505	9673491
0	0.16	178560	178600
0	0.2	711215	710705
0	0.1	558050	557053
0	0.18	1709088	1700424
0	0.16	789792	787493
0	0.16	194085	193993
0	0.23	4320228	3919284
0	0.16	478823	476291
0	0.18	2681154	2668109
0	0.18	656327	612656
0	0.28	240665	243343
0	0.18	1188788	1186913
0	0.03	127887158	127737436
0	0.16	158710195	158650852
0	0.18	176082615	176305902
0	0.09	39686269	39601305
0	0.26	90013199	90011228
0	0.17	19779465	19768542
0	0.16	26272257	26262977
0	0.18	25345957	25353866
0	0.11	5222716	5222589
0	0.26	50863537	50856328
0	0.18	8594914	8565831
0	0.19	371947	368460
0	0.09	9797398	9780669
0	0.27	18119440	18115709
0	0.15	9199002	9081073
0	0.28	12119141	12034603
0	0.07	1514188	1512356
0	0.25	848972	848934
0	0.2	1599963	1600948
0	0.13	613064	613154
0	0.09	1672164	1671940

Raw 2Way Issue Data

Benchmark	SBTOTAL	SBHINTED	DINSTOT	DINSNEW	DBPOLD	DBPNEW
a2time01	222	60	44504	44962	8849	8042
aifftr01	203	57	5205583	5260432	709833	261611
aifirf01	196	52	72944	72854	11833	9447
aiifft01	193	51	5009191	5019180	686495	245058
basefp01	170	39	60652	60720	11208	8643
bitmnp01	353	70	370420	369364	60674	37541
cacheb01	187	34	239070	238298	36590	17397
canrdr01	237	46	1122125	1104622	268432	96581
idctrn01	277	45	409551	408112	50249	31729
iirflt01	240	89	65331	64285	10938	9274
matrix01	303	54	2371357	2371196	452385	222633
pntrch01	201	54	248200	248526	57897	23381
puwmod01	255	36	1790167	1771864	431212	153336
rspeed01	168	37	401152	399597	95399	38381
tblook01	186	56	114006	113313	25555	19378
ttsprk01	305	36	761984	759302	181275	69273
cjpeg	1605	499	66469032	66321206	7343149	1005976
djpeg	1871	607	93190794	92873152	14192112	6897377
rgbcmy	157	39	119966124	120706098	27856479	7556304
rgbhpg	135	27	25493493	25461427	3515694	1081716
rgbyiq	154	36	60507041	60212497	10097001	6415242
ip_pktcheck	1032	64	12797880	12953787	2994032	925694
ip_reassembly	1159	142	15355667	15499895	3239834	991001
nat	2197	283	15297396	15363066	3651347	1041810
ospfv2	1018	64	3194773	3117231	907792	148083
qos	1957	249	26490998	26486538	6187306	2732128
routelookup	1011	60	5738990	5721649	1422391	578652
tcp	1402	170	140520	139592	20764	18472
bezier01	145	26	4698619	4689751	571529	250672
dither01	151	28	11556774	11538718	1911041	1343649
rotate01	219	97	4694874	4637594	1271293	377792
text01	340	113	6778734	6697199	1683939	906053
autcor00	143	29	1112918	1111536	104138	55014
conven00	144	27	506602	504220	85285	61002
fbital00	155	32	975966	974604	148204	74160
fft00	170	39	409454	409385	52102	31325
viterb00	159	29	991754	990498	94455	46651

Raw 2Way Issue Data

PWROLD	PWRIDEAL	PWRNEW	INSCOMMIT	INSCOMMITN	B_PRW_OLD
15.54	14.59	15.37	41111	41743	0.45
11.02	10.39	10.59	4814733	4855993	0.67
14.39	13.58	14.14	68519	68473	0.46
10.98	10.36	10.55	4618141	4616553	0.67
14.92	14	14.59	57251	57249	0.47
11.63	10.87	11.33	340175	339391	0.68
11.94	11.26	11.51	219942	219942	0.65
9.68	8.36	8.67	958477	948313	1.21
11.69	10.98	11.22	401710	400715	0.63
14.76	13.97	14.61	61001	60389	0.42
9.62	8.55	9.03	2337752	2337697	1.33
11.59	10.41	10.78	225082	226308	1
9.58	8.36	8.61	1526642	1516506	1.23
10.1	8.9	9.26	344117	344117	1.11
11.9	10.77	11.58	99968	99966	0.79
9.78	8.59	8.94	649657	650289	1.16
10.84	10.22	10.25	64219226	64220119	0.6
10.41	9.7	9.95	87016368	87017396	0.82
9.54	8.39	8.62	103180019	103180817	1.24
9.57	9.01	9.11	23620414	23620160	0.78
9.35	8.65	9.04	54873058	54872393	0.9
9.96	8.75	9.07	11150829	11236010	1.24
11.01	10.05	10.28	13696200	13782323	0.99
10.23	9.05	9.34	13465586	13463832	1.14
10.93	9.4	9.52	2907406	2821385	1.43
13.33	12.06	12.54	24689703	24689714	0.91
9.84	8.57	9.02	4866420	4865521	1.27
14.37	13.65	14.27	133700	133168	0.49
10.01	9.5	9.71	4329290	4327551	0.56
9.81	8.93	9.51	10630031	10629150	0.99
9.76	8.54	8.79	4006281	4006281	1.27
11.04	9.81	10.43	6056241	6055465	1.13
10.82	10.29	10.49	1098257	1097120	0.48
10.2	9.45	9.97	478143	477235	0.8
10.33	9.39	9.84	962087	961353	0.9
10.92	10.31	10.65	388473	388507	0.57
10.1	9.61	9.79	950920	950922	0.53

Raw 2Way Issue Data

B_PWR_IDE	B_PWR_NEW	CYCLES	CYCLESN
0	0.39	71376	71915
0	0.27	4041740	4043002
0	0.35	95840	95784
0	0.27	3880251	3843471
0	0.34	89186	89331
0	0.44	329698	326860
0	0.3	211971	209992
0	0.46	823887	783612
0	0.38	342448	340381
0	0.34	96451	95062
0	0.62	1500413	1489456
0	0.39	218762	217285
0	0.46	1292703	1244917
0	0.45	318157	312766
0	0.6	119171	118281
0	0.45	577815	567870
0	0.07	49730502	49349504
0	0.39	67477369	66895964
0	0.4	83840665	82584280
0	0.24	17237911	17087620
0	0.58	41926883	41442500
0	0.45	9224348	9148293
0	0.36	12388318	12260740
0	0.39	11899119	11700035
0	0.28	2304865	2207585
0	0.42	26467601	26446974
0	0.53	4137572	4067355
0	0.44	165978	164500
0	0.26	3943089	3911980
0	0.67	7577970	7550564
0	0.46	3259254	3136415
0	0.67	5424935	5234339
0	0.24	965463	963419
0	0.58	376917	373948
0	0.44	732553	731000
0	0.36	352025	350015
0	0.22	686459	681172

Raw 16Way Issue Data

Benchmark	SBTOTAL	SBHINTED	DINSTOT	DINSNEW	DBPOLD	DBPNEW
a2time01	222	63	51950	51398	11502	10254
aifftr01	203	56	6078136	6013854	1227218	741816
aifirf01	196	52	82162	81690	15302	12661
aiifft01	193	50	5874591	5775718	1203039	671495
basefp01	170	39	68481	68601	14110	11196
bitmnp01	353	69	429717	422865	89937	61351
cacheb01	187	33	279277	277123	59855	38598
canrdr01	237	45	1467631	1434412	488506	288365
idctrn01	277	47	426948	424409	57104	36668
iirflt01	240	90	74280	73263	14007	12280
matrix01	303	54	2466673	2466651	505432	248314
pntrch01	201	53	296619	292201	86611	48075
puwmod01	255	35	2345199	2292475	788757	461872
rspeed01	168	35	521206	525444	170707	133059
tblook01	186	56	143196	144060	41959	34516
ttsprk01	305	35	1004257	982956	333788	200061
cjpeg	1605	497	69674771	68877679	9423886	1548797
djpeg	1871	606	107151269	107413955	22766481	13830107
rgbcmy	157	38	155392686	150210104	50451231	26696394
rgbhpg	135	26	29866460	29556962	6263206	3417625
rgbyiq	154	37	73870147	72580408	18457045	10298440
ip_pktcheck	1032	62	16184279	16294369	5167679	3776602
ip_reassembly	1159	143	19150870	18532531	5652637	2878826
nat	2197	282	19807927	18884723	6384975	3260560
ospfv2	1018	63	3712293	3633207	1294763	462790
qos	1957	249	33067877	32715306	10012875	3606394
routelookup	1011	59	7592685	7642162	2629466	1858342
tcp	1402	170	153034	151569	25471	22918
bezier01	145	25	5310723	5244846	964004	588293
dither01	151	28	13021238	13027801	2615539	1842582
rotate01	219	97	5467524	5227597	2020320	589700
text01	340	113	8147847	7931551	2498038	1548998
autcor00	143	28	673230	670708	63472	13612
conven00	144	29	623569	624730	135826	84649
fbital00	155	31	996704	995188	159707	83901
fft00	170	38	543715	539226	85544	62174
viterb00	159	29	1049414	1048032	108734	55190

Raw 16Way Issue Data

PWROLD	PWRIDEAL	PWRNEW	INSCOMMIT	INSCOMMITN	B_PRW_OLD
27.29	26.48	27.18	41093	41093	0.65
13.72	13.07	13.36	4814542	4812697	1.5
23.23	22.54	23.24	68519	68473	0.71
13.6	12.95	13.51	4616073	4618082	1.51
25.89	25.09	25.51	57251	57251	0.69
15.82	15.09	15.76	340285	339345	1.22
15.81	15.11	15.48	219942	219942	1.28
12.79	11.67	12.17	958477	958477	2.33
15.68	15.03	15.42	401655	400825	1.12
25.3	24.62	25.28	60947	61014	0.61
11.95	10.93	11.42	2337645	2337807	2.75
15.73	14.59	15.03	226258	225450	1.85
12.61	11.36	11.96	1526010	1526010	2.39
13.7	12.58	13.28	344117	344117	2.08
17.68	16.64	17.35	99968	99966	1.36
13.08	11.99	12.48	649657	649657	2.23
13.58	12.98	13.02	64219272	64220175	1.32
13.08	12.36	12.57	87015476	87016370	1.75
12.35	11.26	12.2	103180254	103179978	2.45
12.7	12.09	12.25	23621032	23620261	1.56
12.37	11.64	11.92	54872316	54870855	1.78
12.85	11.71	12.29	11149935	11167586	2.41
14.33	13.36	14.12	13696275	13695441	2.11
13.46	12.36	13.35	13483125	13464733	2.21
13.4	12.07	12.64	2822279	2821558	2.69
18.19	17.02	16.65	24689714	24689705	1.63
12.71	11.54	12.11	4865641	4866420	2.47
19.91	19.22	19.88	133174	133220	0.93
14.02	13.47	13.67	4329235	4328560	1.11
13	12.18	12.67	10630029	10630031	1.79
12.64	11.46	11.69	4006281	4006281	2.5
14.03	12.93	13.29	6055623	6055388	2.11
15.51	15.05	15.09	655777	654562	0.74
13.86	13.18	13.46	478134	477295	1.4
14.82	14.02	14.38	962149	961353	1.32
14.88	14.3	14.84	477936	477963	1.07
13.25	12.8	12.95	950920	950858	1.01

Raw 16Way Issue Data

B_PWR_IDE	B_PWR_NEW	CYCLES	CYCLESN
0	0.57	55261	54596
0	0.87	2417727	2355222
0	0.55	68733	69054
0	0.88	2326835	2300476
0	0.5	66869	66972
0	0.87	225774	223328
0	0.78	138162	135655
0	1.33	589700	559954
0	0.68	202737	201952
0	0.51	72203	71651
0	1.27	765675	761706
0	0.96	143834	139660
0	1.35	923119	873365
0	1.56	230276	225447
0	1.07	88633	87825
0	1.29	417399	397498
0	0.19	25576017	25088122
0	0.94	39925075	39134499
0	1.36	58508341	57250211
0	0.78	11584552	11226897
0	0.92	29324101	28297600
0	1.7	6315263	6135972
0	1.17	7742677	7494768
0	1.21	8283728	8021543
0	0.95	1450531	1421548
0	0.54	19604771	18596665
0	1.65	2981327	2911761
0	0.87	94207	92995
0	0.67	2594533	2519751
0	1.21	4976926	4941027
0	0.82	2077134	1928066
0	1.41	3501841	3255286
0	0.11	361618	360437
0	0.86	270047	267394
0	0.66	512098	510503
0	0.81	265065	263993
0	0.41	382439	380818

Raw Fixed Bias Data

Benchmark	SBTOTAL	SBHINTED	DINSTOT	DINSNEW	DBPOLD	DBPNEW
a2time01	222	58	51987	51624	11502	10395
aifftr01	203	59	6073211	6245699	1225405	684607
aifirf01	196	51	82162	81906	15302	12592
aiifft01	193	52	5875297	6049734	1203191	675325
basefp01	170	37	68481	68403	14110	11327
bitmnp01	353	75	429717	435002	89937	60639
cacheb01	187	32	279277	284874	59855	36837
canrdr01	237	41	1467631	1514003	488506	269682
idctrn01	277	71	426874	439042	57093	19883
iirflt01	240	75	72572	71456	13420	11834
matrix01	303	55	2466752	2548626	505438	128599
pntrch01	201	54	296688	303483	86647	44328
puwmod01	255	34	2345977	2438691	788898	434596
rspeed01	168	34	521206	527393	170707	94866
tblook01	186	53	143196	146852	41959	27618
ttsprk01	305	34	1004257	1042464	333788	189430
cjpeg	1605	452	69676078	69530480	9424234	1964536
djpeg	1871	563	107152476	108802371	22766756	9783290
rgbcmy	157	33	155393345	161372796	50451354	27629126
rgbhpg	135	25	29865255	30415643	6262914	3138967
rgbyiq	154	32	73871313	75318239	18457381	9418021
ip_pktcheck	1032	60	16185208	16784198	5167891	2742547
ip_reassembly	1159	132	19151811	19532505	5652789	2984282
nat	2197	278	19785458	20122156	6381108	3316697
ospfv2	1018	68	3712302	3835241	1294770	465370
qos	1957	235	33067860	30214607	10012858	6169910
routelookup	1011	59	7592663	7846541	2629433	1381707
tcp	1402	145	153213	152608	25531	23149
bezier01	145	24	5310737	5390270	964001	563258
dither01	151	30	13020067	13312745	2615212	988126
rotate01	219	96	5466390	5385007	2020010	550375
text01	340	92	8146411	8173931	2497653	1627628
autcor00	143	27	742308	740316	72412	22673
conven00	144	27	623582	633740	135826	76292
fbital00	155	33	995511	1062465	159395	20890
fft00	170	39	619933	632805	93407	65893
viterb00	159	30	1049406	1048519	108743	53815

Raw Fixed Bias Data

PWROLD	PWRIDEAL	PWRNEW	INSCOMMIT	INSCOMMITN	B_PRW_OLD
27.27	26.46	27.13	41111	41111	0.64
13.68	13.03	13.3	4815140	4814568	1.5
23.22	22.49	23.16	68519	68473	0.71
13.6	12.95	13.22	4616717	4618061	1.51
25.89	25.09	25.56	57251	57249	0.69
15.82	15.09	15.52	340285	339345	1.22
15.81	15.11	15.44	219942	219942	1.28
12.79	11.67	12.04	958477	948313	2.33
15.67	15.02	15.23	401545	400715	1.12
25.41	24.74	25.4	59546	59605	0.61
11.95	10.93	11.44	2337755	2337755	2.75
15.74	14.59	14.95	226320	225404	1.85
12.62	11.49	12.06	1526642	1526642	2.39
13.7	12.01	12.57	344117	333981	2.08
17.68	16.64	17.17	99968	99966	1.36
13.08	11.98	12.56	649657	650289	2.23
13.58	12.98	13	64220175	64219454	1.32
13.08	12.36	12.51	87016370	87015583	1.75
12.35	11.26	11.82	103180817	103180817	2.45
12.7	12.09	12.26	23619888	23620361	1.56
12.37	11.64	11.94	54873167	54872549	1.78
12.85	11.71	12.17	11150784	11150743	2.41
14.33	13.36	13.69	13697107	13695700	2.11
13.45	12.36	12.88	13463657	13464637	2.21
13.4	12.07	12.33	2822277	2821506	2.69
18.19	17.02	17.1	24689661	24689760	1.63
12.71	11.54	12.06	4865555	4866420	2.47
19.89	19.22	19.85	133174	133168	0.93
14.02	13.47	13.67	4329290	4328565	1.11
13	12.18	12.43	10629212	10629349	1.79
12.64	11.46	11.75	4005502	4006226	2.5
14.03	12.93	13.43	6054506	6056131	2.11
15.31	14.84	14.89	719886	718022	0.77
13.87	13.18	13.43	478189	478079	1.4
14.83	14.02	13.77	961298	961312	1.32
14.67	14.09	14.41	551613	551702	1.07
13.25	12.8	12.93	950865	950867	1.01

Raw Fixed Bias Data

B_PWR_IDE	B_PWR_NEW	CYCLES	CYCLESN
0	0.57	55221	54680
0	0.7	2414915	2463884
0	0.54	68733	68970
0	0.71	2327194	2377281
0	0.51	66869	66921
0	0.77	225774	227014
0	0.68	138162	139435
0	1.08	589700	585212
0	0.3	202509	212410
0	0.51	70859	70226
0	0.56	765865	829970
0	0.78	143961	145386
0	1.07	924014	943364
0	1.04	230276	215954
0	0.79	88633	89539
0	1.04	417399	426484
0	0.21	25576614	25293250
0	0.58	39925577	39948190
0	1.09	58508620	59990200
0	0.63	11583505	11609890
0	0.74	29324550	29418012
0	1.05	6316034	6441633
0	0.98	7743195	7693337
0	0.99	8270567	8277239
0	0.74	1450556	1474343
0	1.09	19604744	16693717
0	1.06	2981862	3036705
0	0.85	94244	93633
0	0.59	2594646	2590469
0	0.53	4976620	5118839
0	0.68	2076138	2008814
0	1.39	3501202	3412484
0	0.19	392452	390952
0	0.73	270105	271033
0	0.11	511736	539756
0	0.72	297403	303331
0	0.4	382376	380374

Raw BTB Resize Data

Benchmark	SBTOTAL	SBHINTED	DINSTOT	DINSNEW	DBPOLD	DBPNEW
a2time01	222	63	44962	44952	8042	8044
aifftr01	203	57	5260432	5192266	261611	304767
aifirf01	196	52	72854	72860	9447	9436
aiifft01	193	51	5019180	5019940	245058	245211
basefp01	170	39	60720	60736	8643	8638
bitmnp01	353	70	369364	369332	37541	37537
cacheb01	187	34	238298	238260	17397	17364
canrdr01	237	46	1104622	1115618	96581	98645
idctrn01	277	46	408112	427313	31729	41136
iirflt01	240	92	64285	63302	9274	9038
matrix01	303	54	2371196	2371292	222633	222642
pntrch01	201	54	248526	247538	23381	23186
puwmod01	255	36	1771864	1771852	153336	153333
rspeed01	168	37	399597	399598	38381	38372
tblook01	186	56	113313	114011	19378	19494
ttsprk01	305	36	759302	759298	69273	69260
cjpeg	1605	500	66321206	66321566	1005976	1005963
djpeg	1871	607	92873152	92872321	6897377	6897209
rgbcmy	157	39	120706098	120705181	7556304	7556111
rgbhpg	135	27	25461427	25462324	1081716	1081926
rgbyiq	154	36	60212497	60213307	6415242	6415423
ip_pktcheck	1032	64	12953787	12980122	925694	928461
ip_reassembly	1159	142	15499895	15526260	991001	993764
nat	2197	283	15363066	15364002	1041810	1041975
ospfv2	1018	64	3117231	3117379	148083	148109
qos	1957	249	26486538	26487781	2732128	2732123
routelookup	1011	60	5721649	5721546	578652	578636
tcp	1402	170	139592	139582	18472	18455
bezier01	145	26	4689751	4689763	250672	250667
dither01	151	28	11538718	11538879	1343649	1343639
rotate01	219	97	4637594	4637534	377792	377772
text01	340	113	6697199	6696832	906053	906026
autcor00	143	28	1111536	1245699	55014	69312
conven00	144	27	504220	504278	61002	61001
fbital00	155	32	974604	974558	74160	74147
fft00	170	39	409385	602340	31325	51892
viterb00	159	29	990498	990450	46651	46639

Raw BTB Resize Data

PWROLD	PWRIDEAL	PWRNEW	INSCOMMIT	INSCOMMITN	B_PRW_OLD
15.37	14.56	15.28	41743	41725	0.39
10.59	10.41	10.62	4855993	4812097	0.27
14.14	13.57	14.08	68473	68473	0.35
10.55	10.35	10.51	4616553	4617358	0.27
14.59	13.98	14.51	57249	57249	0.34
11.33	10.86	11.27	339391	339331	0.44
11.51	11.26	11.49	219942	219942	0.3
8.67	8.35	8.74	948313	958477	0.46
11.22	10.96	11.24	400715	408178	0.38
14.61	14.04	14.6	60389	59605	0.34
9.03	8.55	8.99	2337697	2337807	0.62
10.78	10.4	10.74	226308	225358	0.39
8.61	8.35	8.57	1516506	1516506	0.46
9.26	8.61	9.22	344117	344117	0.45
11.58	10.75	11.52	99966	100598	0.6
8.94	8.58	8.89	650289	650289	0.45
10.25	10.22	10.25	64220119	64220217	0.07
9.95	9.7	9.92	87017396	87016315	0.39
8.62	8.38	8.58	103180817	103179973	0.4
9.11	9.01	9.09	23620160	23620940	0.24
9.04	8.65	9	54872393	54873114	0.58
9.07	8.75	9.03	11236010	11261933	0.45
10.28	10.04	10.24	13782323	13808246	0.36
9.34	9.04	9.29	13463832	13464767	0.39
9.52	9.4	9.49	2821385	2821546	0.28
12.54	12.05	12.48	24689714	24689760	0.42
9.02	8.56	8.96	4865521	4865457	0.53
14.27	13.64	14.21	133168	133168	0.44
9.71	9.48	9.67	4327551	4327551	0.26
9.51	8.93	9.46	10629150	10629304	0.67
8.79	8.53	8.76	4006281	4006217	0.46
10.43	9.8	10.36	6055465	6055040	0.67
10.49	10.25	10.46	1097120	1227511	0.24
9.97	9.45	9.93	477235	477290	0.58
9.84	9.39	9.79	961353	961298	0.44
10.65	10.12	10.51	388507	576932	0.36
9.79	9.61	9.78	950922	950858	0.22

Raw BTB Resize Data

B_PWR_IDE	B_PWR_NEW	CYCLES	CYCLESN
0	0.36	71915	72033
0	0.27	4043002	3999929
0	0.32	95784	95834
0	0.24	3843471	3844466
0	0.31	89331	89405
0	0.4	326860	326718
0	0.27	209992	210035
0	0.41	783612	805464
0	0.41	340381	363007
0	0.31	95062	94153
0	0.56	1489456	1489842
0	0.36	217285	216537
0	0.42	1244917	1244916
0	0.41	312766	312843
0	0.54	118281	119422
0	0.41	567870	567927
0	0.07	49349504	49350358
0	0.35	66895964	66895749
0	0.36	82584280	82583827
0	0.22	17087620	17088202
0	0.53	41442500	41443168
0	0.41	9148293	9169065
0	0.32	12260740	12281235
0	0.35	11700035	11701031
0	0.26	2207585	2207309
0	0.38	26446974	26449816
0	0.48	4067355	4067375
0	0.4	164500	164637
0	0.24	3911980	3912047
0	0.61	7550564	7550833
0	0.42	3136415	3136278
0	0.61	5234339	5233534
0	0.25	963419	1076183
0	0.53	373948	374065
0	0.4	731000	730982
0	0.4	350015	495221
0	0.2	681172	681155