Worst-Case Execution Time Analysis for Dynamic Branch Predictors

Submitted for the degree of Doctor of Philosophy

Ralf Dieter Reutemann

Department of Computer Science,
University of York

January 2008
Abstract

Microprocessors are being introduced to an increasing number of applications in the domain of safety-critical real-time systems. The analysis of Worst-Case Execution Times (WCET) is essential in order to guarantee that tasks meet their timing constraints. This requires a high degree of predictability of the temporal behaviour of a system. Recent microprocessors, however, make use of architectural features that trade a higher average processing performance for a lower predictability of execution time behaviour and potentially higher worst-case execution times. Features such as instruction pipelining, out-of-order execution, caching, and branch prediction are key implementation techniques to achieve high microprocessor performance. Branch prediction, which is the main area of interest in this work, is a technique supporting speculative execution, which is the execution of instructions across a branch before the outcome of the branch is actually known. However, for developers of real-time systems, branch prediction present predictability problems. The reasons are they increase the difficulty in analysing software for its WCET without introducing unmanageable pessimism and they increase the variability of the execution time. Although a number of methods to support static WCET analysis have emerged in recent years, the effects of dynamic branch prediction on static WCET analysis have received less attention compared to other micro-architectural features.

The aim of this thesis is to establish a static WCET analysis approach for dynamic branch prediction schemes and to assess the accuracy and efficiency of the approach proposed. An evaluation framework is established in order to analyse the branch prediction behaviour of several example programs and identify typical branch execution patterns. The evaluation results highlight the fact that WCET analysis simply assuming a branch misprediction for each branch results in a significant level of pessimism because actual average-case misprediction rates are often below 10%. Therefore, accurate static analysis of dynamic branch
predictors is paramount to calculating tight WCET estimates.

The static WCET analysis method presented in this work extends, and corrects, previously published branch prediction analysis methods by establishing a classification model that is taking into account the semantic context of the branches in the source code. In terms of static analysis, branches are classified as either being easy-to-predict or hard-to-predict, depending on whether or not their execution behaviour can be determined from their semantic context at compile-time. Effects to be taken into account when integrating the analysis approach with instruction pipeline analysis are discussed and a combined analysis method is proposed. It is shown through a number of case studies using different calculation methods that both accurate and efficient static WCET analysis can be performed for bimodal branch predictors using this classification scheme.

The analysis approach is also extended to global-history two-level branch prediction schemes using a previously published example with the benefit that a more detailed explanation of its results is obtained and the complexity of the method is reduced for branches classified as easy-to-predict. Nevertheless, it is concluded that global-history predictors, although analysis is feasible, are less suited than, for example, bimodal and static branch predictors. This is because the slight average-case performance gain usually achieved by two-level predictors does not justify the additional complexity and pessimism introduced to WCET analysis that is required to model such branch predictors. It may even be possible that the performance gain is outweighed by the additional pessimism caused by the wider scope of analysis.

Finally, various coding and compilation techniques are discussed that are intended to reduce the remaining pessimism for branch instructions classified as being hard-to-predict. The additional benefit of applying these techniques is that the number of misprediction can be reduced in order to improve the overall software performance.
Contents

1 Introduction ............................................. 21
   1.1 Safety-Critical Real-Time Systems ................. 22
   1.2 Worst-Case Execution Time Analysis ............... 24
   1.3 Microprocessors and WCET Analysis ............... 26
   1.4 Thesis Overview and Context ....................... 29
   1.5 Thesis Proposition ................................ 31
   1.6 Thesis Contributions .............................. 32
   1.7 Organisation of this Thesis ....................... 33

2 Literature Review and Background .................. 37
   2.1 Worst-Case Execution Time Analysis ............... 37
      2.1.1 Industrial Practice in General ............... 38
      2.1.2 Applications in the European Space Industry 40
      2.1.3 Static Execution Time Analysis ............... 41
   2.2 Overview of Branch Prediction Techniques ........ 46
      2.2.1 Static Branch Prediction ....................... 49
      2.2.2 Dynamic Branch Prediction ..................... 50
2.3 Low-Level Timing Analysis Techniques
   2.3.1 Instruction Pipeline Modelling
   2.3.2 Instruction Cache Analysis
   2.3.3 Data Cache Analysis and Virtual-Address Caches
   2.3.4 Analysis of Dynamic Branch Prediction Techniques
   2.4 Summary

3 Empirical Evaluation of Branch Predictors
   3.1 Introduction
   3.2 Simulation Environment
   3.3 Sample Programs
   3.4 Distributions of Branch Mispredictions
   3.5 Branch Classification Schemes
      3.5.1 Static Branch Classes
      3.5.2 Repeating Patterns
   3.6 Evaluation of Potential Pessimism
   3.7 Summary

4 Static Analysis of Bimodal Branch Predictors
   4.1 Basic Terminology
      4.1.1 Basic Block and Control Flow Graph
      4.1.2 Static and Dynamic Instructions
      4.1.3 Worst-Case Execution Time for Basic Blocks
      4.1.4 Basic Assumptions
5 Integration with Pipeline Analysis

5.1 Overview of Instruction Pipelining

5.1.1 Basic Concepts

5.1.2 Pipeline Hazards

5.1.3 Instruction Issue Policy

5.2 Modelling Instruction Pipelines

5.2.1 Overlap Between Basic Blocks

5.2.2 Including the Effects of Mispredictions

5.3 Case Study: Tree-Based WCET Analysis

5.4 Program Flow Analysis

5.4.1 Analysis of Loop Statements

5.4.2 Analysis of Conditional Statements

5.4.3 WCET Calculation

5.5 Case Study: ILP-based WCET Analysis

5.6 Summary

6 Global-History Branch Predictors

6.1 Branch Patterns

6.2 Analysis of Simple Branch Patterns

6.2.1 Deriving Branch History Patterns

6.2.2 Bounding the Number of Mispredictions

6.2.3 Experimental Evaluation

6.3 Analysing Loops with Conditional Statements

6.4 Summary
7 Coding and Compilation Techniques Supporting Static Analysis

7.1 Predicated Execution

7.1.1 Instruction-Level Predication Support

7.1.2 Software Predication

7.2 Unrolling of Loops

7.3 Case Study: bladeenc Sample Program

7.4 Summary

8 Conclusions and Future Directions

8.1 Summary of Achievements

8.2 Concluding Remarks

8.3 Future Directions

8.3.1 Integration With Cache Analysis

8.3.2 Branch Interference

8.3.3 Modelling Indirect Branches

A Mispredictions per Static Class

B Mispredictions per Branch Type

C Listing for Timing Analysis Case Study

D Listings for bladeenc Case Study

Bibliography
List of Figures

1.1 Worst-case execution time (WCET) ........................................ 25
1.2 Overall WCET analysis framework ....................................... 29

2.1 Loop inversion applied to *while*-loop ................................. 42
2.2 Pipeline hazard after a *taken* branch ................................. 47
2.3 Two-bit counter branch prediction schemes ............................ 51
2.4 Implementation of a two-level adaptive predictor (GA-g) ............ 54
2.5 A local-history two-level adaptive predictor (PA-p) ................... 55
2.6 Example of a reservation table ......................................... 59
2.7 Execution time effects of pipeline overlap ........................... 61
2.8 Conditional statement example ....................................... 71

3.1 Simulation environment .................................................. 81
3.2 Box-plots of mispredictions per branch predictor ................... 92
3.3 Box-plots of mispredictions per static class (bladeenc) ............. 96
3.4 Box-plots of mispredictions per branch type (bladeenc) ............ 100

4.1 Determining the performance of branch predictors ................ 107
4.2 Example of a loop construct with break statement .................. 111
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>States and transitions for a two-bit saturating counter</td>
<td>117</td>
</tr>
<tr>
<td>4.4</td>
<td>DFA accepting strings with worst-case predictor behaviour</td>
<td>124</td>
</tr>
<tr>
<td>4.5</td>
<td>Control flow graph for a simple loop construct</td>
<td>128</td>
</tr>
<tr>
<td>4.6</td>
<td>Control flow graph of a loop with embedded conditional construct</td>
<td>137</td>
</tr>
<tr>
<td>4.7</td>
<td>Cascaded conditional construct</td>
<td>146</td>
</tr>
<tr>
<td>4.8</td>
<td>A <code>switch</code>-statement translated into assembler code</td>
<td>147</td>
</tr>
<tr>
<td>4.9</td>
<td>Conditional statement embedded within loop</td>
<td>152</td>
</tr>
<tr>
<td>4.10</td>
<td>Estimated execution time of a conditional statement</td>
<td>157</td>
</tr>
<tr>
<td>5.1</td>
<td>Principles of instruction execution</td>
<td>168</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of a pipeline stall</td>
<td>170</td>
</tr>
<tr>
<td>5.3</td>
<td>Timing analysis for a conditional loop construct</td>
<td>175</td>
</tr>
<tr>
<td>5.4</td>
<td>Applying a bimodal predictor to a conditional statement</td>
<td>184</td>
</tr>
<tr>
<td>5.5</td>
<td>Execution time behaviour depending on misprediction penalty</td>
<td>186</td>
</tr>
<tr>
<td>5.6</td>
<td>Annotated control flow graph</td>
<td>187</td>
</tr>
<tr>
<td>6.1</td>
<td>Global history two-level adaptive predictor (GAg)</td>
<td>194</td>
</tr>
<tr>
<td>6.2</td>
<td>Control flow graph for a nested loop construct</td>
<td>196</td>
</tr>
<tr>
<td>6.3</td>
<td>Simulation results</td>
<td>204</td>
</tr>
<tr>
<td>7.1</td>
<td>Predicated ARM assembler code for <code>while</code>-loop construct</td>
<td>216</td>
</tr>
<tr>
<td>7.2</td>
<td><code>while</code>-loop construct with software predication</td>
<td>217</td>
</tr>
<tr>
<td>7.3</td>
<td>Loop construct demonstrating the effect of loop unrolling</td>
<td>220</td>
</tr>
<tr>
<td>A.1</td>
<td>Box-plots of mispredictions per static class (gcc-opt)</td>
<td>238</td>
</tr>
</tbody>
</table>
A.2 Box-plots of mispredictions per static class (gcc) . . . . . . . . . . 239
A.3 Box-plots of mispredictions per static class (bzip2) . . . . . . . . . 240
B.1 Box-plots of mispredictions per branch type (gcc-opt) . . . . . . . 242
B.2 Box-plots of mispredictions per branch type (gcc) . . . . . . . . . 243
B.3 Box-plots of mispredictions per branch type (bzip2) . . . . . . . . 244
C.1 Assembler code for timing analysis case study . . . . . . . . . . . 248
C.2 Assembler code for timing analysis case study (cont.) . . . . . . . 249
List of Tables

3.1 Simulated branch predictor configurations . . . . . . . . . . . . . 84
3.2 Description of the sample programs . . . . . . . . . . . . . . . . 86
3.3 Static branch instruction distribution . . . . . . . . . . . . . . . 86
3.4 Execution frequency of instructions . . . . . . . . . . . . . . . . 87
3.5 Dynamic distribution of instruction classes . . . . . . . . . . . . 88
3.6 Branch direction distributions . . . . . . . . . . . . . . . . . . . 89
3.7 Static branch misprediction rate . . . . . . . . . . . . . . . . . . 90
3.8 Dynamic branch misprediction rate . . . . . . . . . . . . . . . . 91
3.9 Static branch classes . . . . . . . . . . . . . . . . . . . . . . . . 95
3.10 Distribution of static branch instructions over branch classes . . 95
3.11 Branch types based on repeating branch execution patterns . . . 97
3.12 Distribution of static branch instructions over branch types . . . 98
3.13 Impact of branch prediction on IPC . . . . . . . . . . . . . . . . 101
4.1 Branch pattern types classified as easy-to-predict . . . . . . . . . 123
4.2 Maximum number of mispredictions for basic patterns . . . . . . 127
4.3 Predictor behaviour for loop pattern, $n = 3$ . . . . . . . . . . . . 129
Acknowledgements

I would like to thank especially Dr Iain Bate, my supervisor, for his guidance, support, and patience during my work on this thesis. I would also like to thank my internal examiner, Dr Neil Audsley, for his support and the initial idea for conducting research in the area of static execution timing analysis for dynamic branch predictors.

The work presented in this thesis was done while being a part-time research student in the Real-Time Systems Research Group, Department of Computer Science, University of York. Finally, I am grateful to EADS Astrium GmbH, Friedrichshafen, Germany, for sponsoring the initial two years of my study.
Declaration

Certain parts of this thesis have appeared in the following previously published conference papers (principle author highlighted):


Chapter 1

Introduction

Microprocessors are being used with increasing frequency in the domain of safety-critical real-time systems. Such systems have emerged in many application domains including aerospace applications, weapon systems, and automotive equipment. Increasing functional complexity and the need for greater flexibility of contemporary systems has lead to an introduction of computer controlled systems into such application domains. These systems put a high demand on the performance of current hardware platforms, and particularly on microprocessors. Leading-edge performance can only be provided by the latest commercial-off-the-shelf (COTS) microprocessors. These microprocessors are being used in a wide range of state-of-the-art system designs because of their low cost-to-performance ratio. However, such microprocessors are typically aimed at widespread industrial usage in the consumer electronics market and, thus, are not designed with the domain specific requirements of safety-critical systems in mind. Consequently, a wide range of factors has to be considered when commercial microprocessors are introduced into the design of a safety-critical real-time system.

Among processor related factors, such as stringent cost constraints or power consumption, a predictable timing behaviour is probably the most important factor that has to be addressed in the context of such systems in addition to
ensuring the functional correctness.

Analysis of advanced micro-architectural features are of particular interest in the design of real-time systems, since these features usually exhibit non-deterministic timing behaviour and complicate the analysis of the timing behaviour of the software and underlying hardware. Examples of such features include instruction and data caches, out-of-order execution instruction pipelines, and static and dynamic branch prediction techniques.

The purpose of this thesis is to look at the suitability of various branch prediction schemes for the specific requirements of (hard) real-time systems. Branch prediction is an attractive approach, applied by most recent microprocessors, to limit the performance penalties caused by conditional branches. However, the behaviour of branch predictors is difficult to predict because the outcome of branches depends on input data, which is usually unknown at compile-time. In Chapter 2, Literature Review and Background, Section 2.2 provides a general introduction to various static and dynamic branch prediction schemes that are used in recent microprocessors.

1.1 Safety-Critical Real-Time Systems

Safety-critical real-time systems require a high level of integrity, because a failure to meet a requirement may lead to a hazardous situation or even to loss of life (Storey, 1996). Thus, the functional correctness of the computational results produced by a microprocessor is an essential requirement. In the domain of real-time systems, the time at which a system provides these results is also of utmost importance. Timing constraints for real-time systems can usually be categorised into two broad classes; failure to meet a hard real-time constraint represents a potentially catastrophic error condition of a system. The digital flight control computer of a fly-by-wire aircraft is an example of a system with hard real-time
constraints. In contrast to hard real-time constraints, a soft real-time constraint is one where failure to meet a deadline can be tolerated. Such failures only represent disturbances in the operation of a system but do not compromise its integrity.

Real-time systems that are integrated, or embedded, into larger engineering systems with interfaces to the physical environment are also referred to as embedded real-time systems. These systems are the main area of interest in this thesis. In addition to the timeliness of the provided results, embedded real-time systems must also be able to react to external stimuli (sporadic events) that may occur simultaneously and with a random distribution. The concept of processes, or tasks, provides a model that captures the concurrent temporal behaviour of the environment. Tasks represent a set of distinct paths of program control that execute concurrently. The required temporal behaviour of a system is usually specified in terms of timing requirements or constraints derived from the physical environment into which the system is embedded. The main problem is to verify that a given set of tasks can be assigned to the available system resources such that all tasks are able to meet their respective deadlines. Burns and Wellings (2001) provide a detailed introduction to real-time systems.

Correct scheduling behaviour is fundamental in the design of real-time systems in general and hard real-time programs in particular. Both static and dynamic scheduling schemes exist to provide a means of assuring correct scheduling behaviour. However, these schemes usually require an estimation, i.e. an approximate knowledge, of the worst-case execution time (WCET) of each task as a prerequisite for performing schedulability analysis of real-time programs. For example, fixed priority dynamic scheduling is a scheme that determines the order of task execution dynamically during run-time according to a fixed priority order.

Rate monotonic scheduling (RMS), for example, allocates fixed priorities to periodic tasks according to the inverse order of their period (Liu and Layland,
1973). This approach assumes that tasks are independent of each other, and the allocation of priorities is optimal if tasks have a deadline equal to their period, so that the task with the shortest period is assigned the highest priority. A fundamental assumption of this scheduling scheme is that the execution time of each task is bounded and known a priori (Balarin et al., 1998).

In the RMS scheme, schedulability of tasks is verified in terms of a condition based on a least upper bound on the microprocessor utilisation, $U_{\text{max}}$. This condition is sufficient, but not necessary, and is given by

$$U_{\text{max}} = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^n - 1),$$

where $n$ is the number of tasks, $(\tau_1, \ldots, \tau_n)$, $C_i$ is the worst case execution time of task $\tau_i$, and $T_i$ is the period of task $\tau_i$.

If this least upper bound is met the set of tasks is schedulable and the RMS priority assignment is feasible. For $n = 2$, the upper bound on the microprocessor utilisation $U_{\text{max}}$ is 82.8%. Note that for large numbers of tasks $n$ this bound tends to $\ln 2$, which is about 69.3%.

1.2 Worst-Case Execution Time Analysis

Estimation of WCET bounds is an important prerequisite in order to gain confidence that each task meets its specified timing constraints, and, in particular, finishes execution before its deadline. The challenge in predicting the WCET of a task is to find a bound that is both tight and safe (Burns et al., 1996).

A tight bound is within a reasonable margin $\epsilon$ of the actual WCET, i.e. $WCET_{\text{est}} \in [WCET_{\text{act}} - \epsilon, WCET_{\text{act}} + \epsilon]$. The margin $\epsilon$ represents the degree of uncertainty of the WCET estimate, where $WCET_{\text{est}}$ and $WCET_{\text{act}}$ are the
1.2. WORST-CASE EXECUTION TIME ANALYSIS

estimated and actual WCET of a single task, respectively. The relationship between actual and estimated WCET is illustrated in Figure 1.1. The WCET is estimated by means of analytical models based on models of the underlying hardware or measured during execution of a program, for example, in the frame of software testing. Naturally, as also shown in the figure, the measured WCET is always less or equal to the actual WCET. Unfortunately, the actual WCET is unknown for all but trivial pieces of source code.

A safe bound, on the other hand, is strictly greater than or equal to the actual WCET, i.e. the WCET must not be underestimated and $WCET_{est} \in [WCET_{act}, \infty)$. The amount of overestimation, which is $\epsilon = WCET_{est} - WCET_{act}$, represents the degree of pessimism of the WCET estimation. Using unsafe WCET bounds in the design of a real-time system may ultimately lead to situations where tasks are in fact no longer able to meet their respective deadlines, although otherwise demonstrated through a schedulability analysis. It should be noted that the notion of WCET assumes uninterrupted execution of a task. Preemption and blocking of tasks has to be addressed by schedulability analysis.

![Worst-case execution time (WCET)](image)

Several techniques for obtaining WCET figures or estimates of real-time programs have emerged over the past two decades since the first publications on static WCET analysis were presented by Shaw (1989) and Puschner and Koza (1989). Since then, WCET analysis has become an active area of research in the domain of real-time systems and a number of analysis approaches have been developed.
CHAPTER 1. INTRODUCTION

by various research groups (Puschner and Burns, 2000). We will further discuss this area of research in Chapter 2, Literature Review and Background. Available techniques for obtaining WCET figures for software can be broadly categorised into the following two classes:

- **Dynamic measuring techniques** are usually based on traditional program testing and measure the execution time of a program when it is executed on the actual hardware platform. Although the measured execution times are accurate, such techniques cannot guarantee that the measured execution time actually represents the worst-case timing behaviour of the program. Thus, the reliability of execution time figures obtained by dynamic measuring techniques depends greatly on the quality of the data used as stimuli.

- **Static analysis techniques** predict the WCET by analysing the execution paths of a program and modelling the timing behaviour of the underlying hardware. Such techniques can ensure safe execution time bounds and therefore allow for a greater level of reliability than dynamic measuring approaches. However, the accuracy of estimates obtained by static analysis depends significantly on the level of detail of the underlying hardware model.

### 1.3 Microprocessors and WCET Analysis

Providing accurate WCET bounds is a challenging problem that is exacerbated by non-deterministic timing properties of many advanced micro-architectural features found in modern commercial microprocessors. Because or despite of this problem, many existing static analysis techniques continue to use simplified models that do not account for such features. This typically results in overly pessimistic WCET estimations and leads to low utilisation of the available processing performance or system resources (e.g. communication busses) in general, and
maybe even to a false decision that a given set of tasks is not schedulable. A good hardware/software co-design, on the other hand, usually requires efficient use of all available system resources. This is particularly true for embedded real-time systems, where cost and power consumption are also essential aspects of system design.

In order to meet their timing constraints real-time systems require a high degree of predictability. In other words, it must be possible to predict the temporal behaviour of the system \textit{a priori}. However, commercially available microprocessors, which are usually not designed with high-integrity real-time systems in mind, exploit micro-architectural features that prove detrimental to the required degree of predictability. With such features, the execution time of an individual instruction is no longer static, but depends on the context (in other words, the surrounding instructions or processor state) in which an instruction is executed.

Advanced micro-architectural features such as instruction pipelining, caching, out-of-order execution, and dynamic branch prediction are key implementation techniques to achieve high microprocessor performance. A number of methods to support static execution time analysis for microprocessors using such features have emerged over the past years.

Branch prediction is a technique supporting speculative execution, which is the execution of instructions across a branch before the outcome of the branch is actually known. Instead of stalling execution until the branch has been resolved, the microprocessor predicts the outcome of the branch as either being \textit{taken} or \textit{not-taken} and continues to fetch instructions along the predicted path. Although branch prediction is an active area of ongoing research in the field of microprocessor architectures in general, the implications of dynamic branch prediction on the worst-case timing behaviour of real-time systems have received less attention until recently.

A simple approach to account for the worst-case timing effects of branch
CHAPTER 1. INTRODUCTION

prediction would be to assume that all branches are mispredicted. With average branch prediction rates of over 90% (Yeh and Patt, 1991) this would, obviously, result in unacceptably pessimistic execution time estimates. This is illustrated by the following example. With an actual misprediction rate of 10% and an average branch frequency of one out of six instructions the assumption that all branches are mispredicted would cause a 129% overestimation of the actual execution time for a simple single-issue microprocessor with a ten-cycle misprediction penalty. Assuming that no other pipeline stalls occur, for example, due to cache misses, all instructions take one clock cycle to execute on a scalar instruction pipeline. Mispredicted branches require additional ten clock cycles in order to flush the instruction pipeline and continue instruction fetch from the correct path. The number of clock cycles per instruction (CPI) for the two scenarios is $CPI_{mp} = 1 + \frac{1}{6} \cdot 10 \cdot 10\% \approx 1.167$ and $CPI_{wc} = 1 + \frac{1}{6} \cdot 10 \cdot 100\% \approx 2.667$. Thus, the overestimation is $\frac{CPI_{wc} - CPI_{mp}}{CPI_{mp}} \cdot 100\% \approx 129\%$. This overestimation would get even worse for superscalar microprocessors, which can issue multiple instructions each clock cycle, because of the higher performance impact of stalling the instruction pipeline.

The above example illustrates that accurate static analysis of dynamic branch prediction schemes is paramount to calculating tight WCET estimates and a relevant area of research in the domain of safety-critical real-time systems. In addition, branch prediction techniques will become a crucial factor for improving the performance of future uniprocessor architectures, surpassing even the limitations imposed by memory systems (Jouppi and Ranganathan, 1997). Usually, only a small number of branch instructions have actually a misprediction rate of 100% (Hennessy and Patterson, 2006), which we will also demonstrate in Chapter 3, Empirical Evaluation of Branch Predictors, by simulating different static and dynamic branch predictor schemes with a set of sample programs. Yet, these few branch instructions with worst-case misprediction rates contribute significantly
to the performance loss due to pipeline stalls caused by mispredictions.

1.4 Thesis Overview and Context

In order to limit the overestimation involving static WCET analysis of branch prediction schemes to a more acceptable value, the aim of this work is to provide tight but safe upper bounds on the number of branch mispredictions that can be expected for basic control statements, such as loops and conditional constructs. In particular branch instructions that appear within loops, and thus are frequently executed, are expected to provide a promising potential for less pessimistic assumptions about branch misprediction penalties. Also, we expect to improve the efficiency of the analysis by excluding those branches that exhibit worst-case branch predictor performance.

Figure 1.2: Overall WCET analysis framework

Figure 1.2 depicts an example of an overall WCET analysis framework consisting of various high-level and low-level analysis modules (Bate, 2001). The branch prediction analysis module, which is the primary focus of the work in this
thesis, uses the path information provided by the program path analysis module to derive the branch predictor behaviour according to the execution pattern of the involved branch instructions. The set of basic blocks and paths is obtained from object code rather than high-level source code in order to allow the analysis modules to take into account code transformations and optimisations performed by the compiler. For the scope of this thesis we use the GNU C Compiler (Gough and Stallman, 2004) for generating the object code from C source code. We do not assume or exclude any particular compiler optimisation settings except a commonly used code transformation technique called loop inversion (Muchnick, 1997). This assumption and its impact on analysis will be revisited in more detail later in this thesis.

In principle, the branch predictor behaviour depends only on the set of paths provided as input and the execution patterns of the branch instructions covered by these paths. Data and instruction cache performance has no direct impact on the branch predictor analysis. Therefore, the first part of this thesis develops a static analysis approach for dynamic branch predictors independently from other analyses. This approach is then integrated in the second part of the thesis with instruction pipeline analysis.

We assume a five-stage instruction pipeline architecture, like the one used in the SimpleScalar simulator (Austin et al., 2002), as baseline for our instruction pipeline model. This architecture does not resemble a real microprocessor implementation but rather provides an abstract model of a RISC microprocessor using a modified MIPS-IV instruction set architecture. Section 3.2 on page 79 gives a more detailed description of the microprocessor model used in this thesis. Modelling of instruction and data cache behaviour has been excluded from the scope of this work.

In the WCET analysis framework shown in Figure 1.2 on the preceding page, the output of the branch prediction analysis module is providing bounds on the
expected number of branch mispredictions for each branch instruction included in the set of paths. This result is finally integrated with the other low-level analysis results, such as data cache, instruction cache and pipeline analysis, in order to estimate the WCET figure for the piece of software subject to static analysis.

1.5 Thesis Proposition

The focus of this thesis is on static program analysis, since only such techniques can guarantee the level of assurance required for the domain of safety-critical real-time systems. However, safe WCET estimates do not come by the very nature of static analysis but by making conservative assumptions about the timing behaviour when modelling the microprocessor hardware. On the other hand, the level of pessimism associated with the analysis approach should be sufficiently low, or accurate, in order not to waste processing resources. Of course, the amount of pessimism that is tolerable for a given application domain and system context needs to be defined and justified – we will discuss this topic in more detail in the summary of Chapter 2, Literature Review and Background.

Although we want to minimise the level of pessimism, the analysis model should not be too complex to handle in terms of computing resources, for example, processing performance. Thus, another requirement on the design of a static WCET analysis model is that it is not only accurate but also efficient. Very often, though, we are required to make a tradeoff between these two criteria.

The following statement summarises the central proposition for this thesis:

*Accurate worst-case execution time (WCET) analysis for dynamic branch predictors can be performed for a large number of branches in a program by analysis tailored by a branch classification scheme based on a theoretical model of branch*
CHAPTER 1. INTRODUCTION

prediction, and the results of the analysis then integrated with other parts of the WCET analysis.

The evaluation of the above thesis proposition will be organised around the following set of criteria:

1. accuracy, efficiency and scalability of the analysis approach;

2. relevance of the analysis for a large number of branch instructions in a program; and

3. modularity in order support integration with other analysis methods.

These criteria will be further elaborated in the summary of Chapter 2, Literature Review and Background, and finally used in Chapter 8, Conclusions and Future Directions, to evaluate the research results presented in this thesis against the above proposition.

1.6 Thesis Contributions

In this thesis, the aim of the research is to establish a static WCET analysis approach for dynamic branch prediction schemes and to assess the accuracy and efficiency of the approach proposed based on various case studies. The analysis approach provides a theoretical model for the behaviour of a bimodal branch predictor using a two-bit saturating counter and derives upper bounds on the number of branch mispredictions for typical control flow graph configurations.

The main contributions of this thesis over the state-of-the-art in the research field of static WCET analysis are summarised in the following:

1. The thesis evaluates the behaviour of various static and dynamic branch prediction schemes (Smith, 1981; Yeh and Patt, 1993) for a number of
sample programs in order to contribute to a better understanding as to how branch predictors work in general.

2. Based on the observations about branch behaviour in this thesis a static analysis method for bimodal branch predictors is defined using a classification model that is based on the static predictability of branch instructions, i.e. branches are classified according to their program context as either being hard-to-predict or easy-to-predict, published previously in Bate and Reutemann (2004). For the latter category of branches, the approach provides accurate upper bounds on the number of branch predictions, while for the remaining branch instructions worst-case branch predictor behaviour is assumed.

3. The integration with instruction pipeline analysis and the static analysis method for bimodal branch prediction has been demonstrated for different WCET calculation methods based on Implicit Path Enumeration Techniques (IPET) and a hierarchical approach extending the original timing schema by Shaw (1989). This approach has been published previously in Bate and Reutemann (2005).

4. Finally, the thesis identifies the difficulties associated with static WCET analysis of dynamic branch prediction schemes that take into account more complex execution patterns, such as global-history two-level predictors, and proposes further areas of research to cope with the additional complexity required for modelling such predictors.

1.7 Organisation of this Thesis

Based on the research objectives and motivations outlined in the previous sections, the remainder of this thesis is structured as follows, together with a brief
description of the contents of each chapter:

- **Chapter 2, Literature Review and Background**, surveys previous work relevant to the subject of this thesis. The first section discusses a number of different analysis techniques for worst case execution time that have emerged in recent years. The focus of this review is on low-level timing analysis, i.e. analysis techniques that take into account the effects on execution time of micro-architectural features such as instruction pipelines, branch prediction, and caches. Further areas of research relevant to the subject of this thesis are motivated from the results and shortcomings of these existing analysis techniques. The chapter also provides some background on static and dynamic branch prediction techniques.

- **Chapter 3, Empirical Evaluation of Branch Predictors**, describes the simulation methodology and metrics used to establish an evaluation framework for branch prediction analysis. The main metric being evaluated in this context is the branch misprediction rate as it is independent from other hardware architectural features. The evaluation framework analyses the branch prediction behaviour of several example programs using static classification schemes that are based on the dynamic behaviour of branch instructions. Based on these results branch instructions are classified as being either easy-to-predict or hard-to-predict. The purpose of this classification is to provide a statement of how many branches are typically easy to predict.

- **Chapter 4, Static Analysis of Bimodal Branch Predictors**, describes the transition from branch classification based on dynamic branch behaviour to a classification approach based on properties of branch instructions that can be determined statically. Then the chapter provides the theoretical foundation and a method for static analysis of bimodal branch prediction schemes. The underlying idea of the method is to transform an existing ap-
approach for data cache analysis to the area of WCET analysis for dynamic branch predictors. Basic control flow statements, such as loops and conditional constructs, of the C programming language are analysed in order to derive an upper bound on the number of mispredictions that can be expected for a bimodal branch predictor.

- **Chapter 5, Integration with Pipeline Analysis**, shows how the static analysis method for branch predictors presented in the previous chapter can be integrated with an instruction pipeline model. Using this integrated low-level analysis model, calculation of the WCET figure is discussed for two different calculation schemes; one based on *Implicit Path Enumeration Techniques* (IPET) and the other following a tree-based analysis approach extending the original timing schema presented by Shaw (1989) with the effects of branch mispredictions.

- **Chapter 6, Global-History Branch Predictors**, extends the method presented in Chapter 4 to take into account more complex two-level branch predictor configurations that use global branch history for predicting the direction of branches. A simple nested loop and a loop containing a conditional construct are analysed to explore the space of possible branch history patterns, which are grouped into different pattern types. Based on these pattern types, upper bounds on the number of mispredictions that can be expected for the two examples are stated.

- **Chapter 7, Coding and Compilation Techniques Supporting Static Analysis**, presents coding and compilation techniques for reducing the pessimism associated with static WCET analysis of branches that remain hard-to-predict. Predicated execution and loop unrolling, two techniques aiming at the elimination or reduction of the number of branches in a program, are introduced and a case study is presented that demonstrates how these
techniques can be applied to one of the sample programs discussed earlier as part of the empirical evaluation of branch prediction schemes in Chapter 3.

- Finally, Chapter 8, *Conclusions and Future Directions*, summarises and evaluates the results of this work and provides suggestions for further avenues of research in the area of static WCET analysis for microprocessors using branch predictors.
Chapter 2

Literature Review and Background

The purpose of this chapter is to survey the literature and provide some technical background relevant to the subject of this thesis. Section 2.1 provides an overview of previous work on WCET analysis in general and describes the current industrial practice using mainly dynamic measurement techniques based on software testing. Since the main focus of this thesis is on static WCET analysis for dynamic branch prediction techniques, Section 2.2 provides some background on such techniques in order to aid the understanding of the discussion of low-level timing analysis techniques in Section 2.3. Finally, Section 2.4 summarises the literature review and motivates further work presented in subsequent chapters of this thesis.

2.1 Worst-Case Execution Time Analysis

Estimation of WCET bounds is an important prerequisite for performing schedulability analysis of real-time tasks. We must assure that each task meets its specified timing constraints, and, in particular, finishes execution before its deadline.

Static execution time analysis methods provide a greater level of assurance than dynamic measuring based on traditional ad hoc testing techniques. This
is because such methods can cover all feasible combinations of input data using program flow analysis and a model of the underlying physical execution environment, most importantly the microprocessor. Exhaustive testing of all possible program paths, on the other hand, is usually not possible due to the significant effort that is required. Some recent approaches suggest to combine static analysis and measurement based analysis in order to overcome the problem of accurately modelling the target hardware.

2.1.1 Industrial Practice in General

Dynamic measuring techniques are frequently used in industry to demonstrate that the timing requirements of a system are met. Typically, a program is executed with a limited set of test cases as stimuli and the WCET is obtained from the longest execution time that has been measured. The measured execution times are usually more accurate than those obtained by static WCET analysis techniques since all timing properties of the underlying hardware are included in the measure without the need to establish detailed analysis models of the hardware. However, dynamic measuring techniques cannot provide safe WCET bounds because it is usually not possible to guarantee that the measured bound actually represents the worst-case temporal behaviour of a program. Although exhaustive testing of all possible (or feasible) program path permutations provides a sufficient solution to overcome this problem, this is not feasible for all but trivial programs. Moreover, the depth of the conducted tests is limited due to cost and resource considerations, and the problem clearly is to select appropriate test data. Consequently, the reliability of measures obtained by such techniques greatly depends on the quality of the input data used as stimuli. Another disadvantage of dynamic measuring is that it can only be used in the later phases of system development when the actual hardware environment is available. Nevertheless, dynamic measuring plays an important role in the verification and
validation of safety-critical real-time systems. The importance of testing in the domain of real-time systems is illustrated by the fact that software projects often spend more than 50% of their overall development budget on testing (Davis, 1979; Beizer, 1990; Harrold, 2000). This is in part because real-time systems also require examination of timing properties in addition to demonstrating functional correctness.

In the context of safety-critical real-time systems missing a deadline can lead to catastrophic consequences and therefore simply relying on testing alone may not be adequate for such systems. Complementing testing by using static WCET analysis early in the software development life-cycle can provide supporting evidence that a system will be able to meet its deadlines.

However, application of static WCET analysis methods supported by tools has been adopted in a limited range of industrial application domains, such as the aerospace and automotive industries. Yet, dynamic timing measurements obtained through extensive software testing are still considered best practice. Sehlberg et al. (2006) present a case study where the aiT WCET analysis tool developed by AbsInt was used to find upper time bounds for task-oriented vehicular control code. With relatively little effort, the analysis tools allowed the developer to produce safe WCET estimates for individual tasks. These estimates are tighter than those usually obtained by measurements with some safety margin added. They further conclude that for the code under study, which includes many infeasible program paths, significant work is necessary to manually exclude such paths from analysis in order to obtain even tighter WCET estimates. Currently, however, the annotation language of the aiT WCET analysis tool does not provide the semantics for expressing mutual exclusiveness of instruction execution.

In a recent work, Gustafsson and Ermedahl (2007) summarise the experience from different industrial case-studies using both static and measurement-based tools. They conclude that static analysis methods can provide a complete view of
the code under analysis and the longest execution path. Dynamic measurement techniques can complement this view by executing the worst-case paths using the required input parameters. This results in tighter overall WCET analysis estimates.

### 2.1.2 Applications in the European Space Industry

The European space industry has recently seen the use of static WCET analysis tools, such as Bound-T (Holsti and Saarinen, 2002), in the frame of the independent software verification and validation (ISVV) activities for the on-board data management and attitude control software of an earth observation satellite (Rodríguez et al., 2003).\(^1\) The hardware platform for this software is based on a rather simple ERC32 32-bit microprocessor architecture based on the SPARC V7. This microprocessor architecture is used in on-board computers of many European space projects and will likely be replaced in the future by the LEON microprocessor (Gaisler, 1999). Unlike the ERC32, the LEON microprocessor overcomes the speed gap between the processor and the memory by using a fast, on-chip cache memory. A study awarded by the European Space Agency (ESA) to a consortium consisting of academic and industrial researchers has investigated the introduction of a cache-equipped microprocessor into space systems (Vardanega et al., 2007).

Despite these research efforts led by the European space community, the systematic use of static analysis tools in the software development process is far from being well established in this application domain.

An overview of static WCET analysis in general and a review of related previous research work is provided in the following section, and later in Section 2.3,

---

\(^1\)This view is drawn on personal experience of the author working in the space industry for almost one decade.
which discusses low-level timing analysis techniques.

2.1.3 Static Execution Time Analysis

In the domain of safety-critical real-time systems, calculating WCET bounds is an important prerequisite in order to gain confidence that each task meets the timing constraints it has to obey, and, in particular, finishes execution before its deadline.

Early research work that addresses the problem of predicting WCET figures using analytical methods is presented by Shaw (1989). His approach provides a set of formulae for specifying and reasoning about timing properties of concurrent programs. Shaw uses logical assertions and extends program proof techniques to derive time bounds for basic high-level language constructs using an approach called timing schema. For example, bounds on the execution time \( t(S) \) of a statement \( S \) can be obtained by

\[
T(S) = [t_{\text{min}}(S), t_{\text{max}}(S)],
\]  

(2.1)

where \( t_{\text{min}}(S) \leq t(S) \leq t_{\text{max}}(S) \) for all statements \( S \) in a program.

The execution time bounds of individual constructs are combined in a bottom-up fashion to obtain bounds for larger program blocks. For example, the timing schema for the sequential composition of two statements \( S = S_1; S_2 \) is given by

\[
T(S) = T(S_1) + T(S_2)
\]  

(2.2)

According to the timing schema, the WCET of a conditional language construct \( S = \text{if } B \text{ then } S_1 \text{ else } S_2 \) is calculated as

\[
T(S) = T(B) + \max(T(S_1), T(S_2))
\]  

(2.3)
The WCET of a loop construct, e.g. a while-loop, is calculated by multiplying the WCET of the loop body with an upper bound on the number of loop iterations (also called loop bound). Then, for a statement $S = \text{while } B \text{ do } S_b$ the WCET is given by

$$T(S) = (n + 1) \cdot T(B) + n \cdot T(S_b),$$

where $n$ is the number of loop iterations.

The while-loop tests a boolean expression $B$ (loop condition) at the top of its loop body and executes the loop body, represented by $S_b$, if the expression $B$ evaluates as being true. Therefore, assuming $n$ loop iterations, the loop condition is evaluated $n + 1$ times, as shown in Equation (2.4).

In contrast to the work done by Shaw (1989), we will present a technique in this thesis that analyses the execution time behaviour on object code level rather than for high-level source code. This is because modern compilers, such as the GNU C Compiler (Gough and Stallman, 2004), apply various optimisations techniques during code generation with the side effect that the control flow graph on object code level may be different from the original source code.

![Figure 2.1: Loop inversion applied to while-loop](image)

Figure 2.1: Loop inversion applied to while-loop

Figure 2.1 illustrates the effect of a loop optimisation technique called loop inversion (Muchnick, 1997). The original while-loop construct, which is shown in Figure 2.1(a), is translated in an equivalent construct where the loop condition
is evaluated at the bottom of the loop body (Figure 2.1(b)). The advantage of this transformation is that the generated object code for the loop body requires only one conditional branch instruction instead of two branches. Some instruction set architectures directly support this type of loops by providing dedicated instructions for decrementing and evaluating a loop counter register.

It is essential that all loop statement have a bounded number of iterations in order to be able to calculate a finite WCET estimate. Loop bounds are either provided as annotations by the programmer or derived automatically by program flow analysis.

The timing schema expresses best-case and worst-case execution times of language constructs as fixed time bounds. In other words, it is assumed that the execution time of statements is fixed and does not vary depending on the program context of a construct. In particular, path information is not used by the timing schema, for example, to take into account that the longer statement in a conditional construct may not always get executed all the time. Ignoring path information for complex program structures may lead to pessimistic WCET estimates by several orders of magnitude.

Like other early static analysis methods Shaw concentrates primarily on high-level language constructs and simply ignores the effects of instruction pipelines on program execution time, which leads to overly pessimistic WCET bounds. It should be noted, however, that at the time of Shaw’s work, instruction pipelines and other complex micro-architectural features either did not yet exist or at least did not contribute significantly to the pessimism associated with static WCET analysis.

Modern microprocessors take advantage of the implicit parallelism in programs and use instruction-level parallelism to improve overall instruction throughput. Since the benefit of overlapping instruction execution (instruction pipelining) or execution of multiple instructions per clock cycle (superscalar instruction
issue) is limited by resource, control, and data dependencies among instructions, the time to execute an instruction now depends on its context. Thus, the applicability of the original timing schema approach proposed by Shaw (1989) for recent microprocessor architectures is quite limited. Nevertheless, the timing schema serves as the common basis of a number of more recent and complex static WCET analysis techniques.

Engblom et al. (1999) partition the problem of static program analysis into the following three main research areas:

1. **Program flow analysis** determines the worst-case, i.e. the longest, execution path through a program. Infeasible program paths, i.e. paths that are never executed independent of the provided input data, can contribute significantly to overestimation of execution times and thus have to be removed from further analysis. Program flow analysis can be carried out either on assembler instructions (object code) or high-level\(^2\) language constructs (Shaw, 1989). In order to obtain a finite value for the WCET it is necessary that an upper bound on the number of loop iterations is known for each loop construct. Such bounds are either provided as manual program annotations by the programmer (Chapman et al., 1994) or are determined automatically (Healy et al., 1998; Martin et al., 1998; Gustafsson et al., 2006; Kazakov and Bate, 2006). For the purpose of this thesis, it is assumed that such information is available from program flow analysis.

2. **Low-level analysis** derives the WCET of basic blocks or individual instructions using a model that addresses the effects of advanced hardware features, such as scalar instruction pipelines (Lim et al., 1995; Hur et al., 1995), superscalar instruction pipelines (Lim et al., 1998; Schneider and Ferdinand, 2006).

\(^2\)When performing program flow analysis at high-level we have to perform checks that the compiler does not alter the control flow, e.g. by introducing additional nodes into the control flow graph in order to perform range checking of variables.
2.1. WORST-CASE EXECUTION TIME ANALYSIS

1999), instruction pipelines with out-of-order execution (Li et al., 2006),
direct-mapped instruction caches (Müller, 1994), associative instruction
 caches (Müller, 2000), data caches (Rawat, 1993; White et al., 1999), or
dynamic branch prediction (Colin and Puaut, 1999; Mitra and Roychoudhury,
2001; Li et al., 2004; Bate and Reutemann, 2004, 2005), on execution
time.

Traditionally, the primary problem of low-level analysis was to determine
the timing effects of pipeline overlap between successive instructions (local
effects analysis). However, the introduction of caches and dynamic branch
prediction techniques requires a broader scope of analysis because the whole
program context must now be taken into account (global effects analysis).
Section 2.3 provides a detailed review of previous work in the area of low-
level timing analysis.

3. Calculation combines the results of program flow analysis and low-level
analysis to calculate the final WCET. Three different calculation methods
have been proposed in the literature: Path-based methods explicitly explore
all feasible execution paths to find the longest path for a scope (for exam-
ple, a function or a loop construct), with a program being split into several
scopes (Healy et al., 1999, 2000). The overall WCET figure is calculated
hierarchically on all program scopes. Tree-based methods generate the final
WCET estimate by a bottom-up traversal of a tree representing the pro-
gram (Shaw, 1989; Puschner and Koza, 1989; Lim et al., 1995; Hur et al.,
1995). Methods based on Implicit Path Enumeration Techniques (IPET)
express the program flow and low-level execution time using arithmetic
constraints. Program execution paths are enumerated implicitly by this
calculation method. This is in contrast to path-based calculation methods,
where all paths are enumerated explicitly. To calculate the final WCET
estimate, Integer Linear Programming (ILP) is used to maximise a goal
function that combines the linear constraints derived from the control flow graph and the modelling of the hardware (Li et al., 1995; Puschner and Schedl, 1995; Li and Malik, 1995; Mitra et al., 2002; Li et al., 2003). A comparison between different WCET analysis methods is presented by Engblom et al. (2000).

While early work usually tries to tackle all of the above three problem areas within an integrated approach, later work takes a modular approach by separating program flow analysis and low-level analysis (Ermedahl, 2003). This allows each analysis method to be optimised for its specific problem domain and low-level analysis methods to be adopted for different hardware environments.

The remainder of this chapter will focus on previously published work in the area of low-level analysis and hardware modelling for advanced micro-architectural features employed by contemporary microprocessors. Since the main focus of this thesis is on static WCET analysis for branch prediction techniques, the following section gives a general introduction to various available techniques in order to aid the understanding of the discussion of low-level analysis related to branch prediction.

### 2.2 Overview of Branch Prediction Techniques

Branches in the control flow of a program severely limit the instruction-level parallelism that can be exploited by a microprocessor. Although superscalar microprocessors are capable of issuing several instructions each clock-cycle, the actual average issue rate for most programs is significantly lower because of various limitations on instruction-level parallelism (Wall, 1993), including the detrimental effects of control flow dependencies. For example, the Alpha 21264 microprocessor has a peak issue rate of six instructions per clock cycle and a sustainable rate of four instructions on either integer or floating-point code (Kessler, 1999).
2.2. OVERVIEW OF BRANCH PREDICTION TECHNIQUES

Figure 2.2: Pipeline hazard after a taken branch

Figure 2.2 shows how a taken branch delays instruction fetching until the outcome of the branch is resolved and the next sequence of instruction can be fetched from the new target location. We assume an instruction pipeline split into the following five stages:

- **Instruction fetch** (IF). Fetches the instruction referenced by the program pointer (PC) from memory.
- **Instruction decode** (ID). Reads the processor registers and decodes the instruction.
- **Execution** (EX). Executes the operation or calculates an address.
- **Memory access** (MEM). Operands in data memory are accessed.
- **Write back** (WB). This stage stores the result of the instruction execution into the register file.

The numbers of cycles required to execute and resolve the outcome of a branch is a function of the pipeline architecture and can have a significant effect on processing performance especially for long pipelines. In this example, it is assumed that the branch target address is available after the MEM stage, which causes a three cycle delay for taken branches. All instructions that are fetched after the branch have to be removed from the pipeline (pipeline flush) and the pipeline is stalled until the outcome of the branch is known, i.e. the result of the condition associated with the branch instruction is available.
Modern microprocessors combine the approach of dynamic instruction scheduling with branch prediction in order to try to alleviate the problem of disrupting the instruction flow into the pipeline due to branches. With branch prediction, the outcome of a conditional branch is predicted as either being taken or not-taken and instructions are fetched along the predicted path. This is called speculative execution because instructions are executed before it is known that they should execute. These instructions are only allowed to retire when the actual (resolved) branch direction has been predicted correctly. Otherwise, all speculated instructions have to be removed from the pipeline. Consequently, all instructions issued after a branch are not allowed to change the machine state until the branch is resolved and the new instruction fetch address is known.

A survey of branch prediction and other techniques to reduce branch effects can be found in Uht et al. (1997). In addition, Hennessy and Patterson (2006) provide a detailed presentation of various static and dynamic branch prediction techniques, and simulate a set of sample programs based on the SPEC benchmark suite to obtain an quantitative assessment of how well branch prediction works in general. Evers (2000) further investigates how branches behave and why they are predictable. An interesting outcome of the work presented by Evers is that two thirds of all branch instructions are very predictable using simple branch predictors because these branches follow repeating patterns.

Two basic categories of branch prediction techniques can be distinguished depending on whether or not past execution history is used for making a prediction about the direction a branch instruction will take in the future (Smith, 1981; Lee and Smith, 1984):

- **static branch prediction**, where the prediction of the branch direction, i.e. taken or not-taken, remains static during execution; and

- **dynamic branch prediction**, where the prediction of the branch direction
may be changed in the course of program execution based on recent branch execution history that is being collected and analysed by the branch predictor.

The principles of these two categories of branch prediction techniques are described in the following two subsections.

2.2.1 Static Branch Prediction

Static branch prediction is a simple prediction technique that uses only static information to determine the predicted direction. Smith (1981) distinguishes between four static branch prediction strategies:

- Always predict that a branch is not-taken (e.g. Intel i486). This strategy assumes a straight flow of fetched instructions.
- Always predict that a branch is taken (e.g. Sun SuperSPARC). This is an advantage for instruction pipeline architectures where the target address is available before the branch outcome.
- Predict the outcome of a branch according to its opcode.
- Predict backward branches taken and forward branches not-taken (e.g. HP PA-7x00).

Early PowerPC microprocessors, for example, support semi-static predictors in their instruction-set architecture. Semi-static branch predictors are a variation of static predictors. The prediction is no longer static for all branch instructions but now depends on prediction information included in the opcode of the branch instruction. The compiler uses program profile statistics to generate the prediction information for each branch instruction. However, the accuracy of these predictors is limited because branch statistics can vary from the sample data to the actual data.
2.2.2 Dynamic Branch Prediction

Dynamic branch prediction techniques base their prediction on recent execution history of a branch. In general, dynamic branch prediction consists of two parts:

- Predicting the branch target address. This is simple for direct jumps or PC-relative branches because the target address does not change during program execution. In this case, caching of the target addresses is sufficient. Indirect jumps or branches are much more difficult to predict.

- Predicting the direction (outcome) of the branch.

Branch Target Buffers

A branch target buffer (BTB) or branch target address cache (BTAC) is a special instruction cache that predicts the address of the next instruction to be fetched after a taken branch occurred. The branch target buffer only stores target addresses for taken branches because in the case of not-taken branches the next sequential instruction is fetched. The implementation of a branch target buffer is effectively identical to caches. Each entry in the buffer represents an association between the address of a taken branch and the target address of the next instruction to fetch. If the address of a fetched branch instruction matches an entry in the branch target buffer instruction fetching continues immediately at the target address. Otherwise, the branch is assumed to be not-taken and the instruction following the branch is fetched from memory. Thus, a branch target buffer predicts that a branch will execute the same as in its previous execution.

Bimodal Branch Predictors

Smith (1981) presents various hardware techniques for branch prediction including bimodal prediction, which is the simplest dynamic branch prediction tech-
2.2. OVERVIEW OF BRANCH PREDICTION TECHNIQUES

A bimodal branch predictor is a simple table of binary values, also called branch history table (BHT), indexed by the lower bits of the address of a branch instruction. In general, each entry in the BHT contains $n$ bits that record state information about the recent execution history of a branch. A one-bit branch predictor simply predicts that a branch will always execute the same way next. A branch misprediction results each time a branch changes its direction, i.e. from taken to not-taken, or vice-versa.

Consider a loop that is executed several times (e.g. with the execution pattern $(TTTN)^n$). A one-bit predictor causes a misprediction for the last iteration of the loop and another one when the loop is entered again. This is avoided by two-bit predictors, which implement a saturating two-bit up/down counter for each entry. Two-bit counters allow for some degree of hysteresis and thus are less affected by occasional changes in branch direction. Bimodal predictors with

---

3RISC microprocessors usually align instruction addresses on four byte boundaries. Therefore, the least significant two bits are not used for selection.

4For the remainder of this thesis, $T$ denotes a taken branch, and $N$ denotes a not-taken branch.
three or more bits do not appear to offer a significant advantage over two-bit predictors (Šilc et al., 1999).

Figure 2.3 on the previous page shows the states and transitions for two different implementations of a two-bit counter branch prediction scheme (bimodal branch predictor). The state of the counter is updated after the branch outcome has been resolved. Each taken branch increments the counter, and each not-taken branch decrements the counter. The prediction is made based on the most significant bit of the predictor’s state: 1 is predict taken, 0 is predict not-taken. Bimodal predictors work best for branches that are biased to either the taken or not-taken path.

Examples of microprocessors that use bimodal predictors include the Alpha 21064 (McLellan, 1993) and 21164 (Edmondson et al., 1995), and the MIPS R10000 (Yeager, 1996). The Intel Pentium (P5) microprocessor includes this predictor in a branch target buffer. Other microprocessors, e.g. the PowerPC, use a BHT for branch direction prediction and a separate BTB for target address prediction. The SUN UltraSPARC-I (Tremblay and O’Connor, 1996) uses the modified counter shown in Figure 2.3(b) on the preceding page. According to Tremblay and O’Connor (1996), simulations on a large set of benchmarks showed that the modified counter scheme improves the branch misprediction rate compared to other algorithms based on the 2-bit counter mechanism. Nevertheless, the UltraSPARC-I microprocessor seems to be the only implementation of this predictor scheme.

All branch prediction schemes based on a $n$-bit saturating counter exhibit worst-case predictor performance (i.e. all branches are mispredicted) when a branch instruction alternates between its taken and not-taken directions, or vice-versa. The most notable difference between the two different counter schemes shown in Figure 2.3 on the previous page is their different worst-case behaviour. The worst-case branch behaviour for the modified counter scheme in Figure 2.3(b)
is when a branch instruction changes its direction every second execution instead of every execution in the case of the saturating counter scheme shown in Figure 2.3(a).

A more detailed model and discussion of bimodal branch predictors and their worst-case behaviour will be provided in Section 4.2 on page 116.

**Two-Level Branch Predictors**

Yeh and Patt (1991) propose a new dynamic branch prediction technique, called *two-level adaptive branch prediction*, that uses two levels of branch history information to make predictions. Two-level adaptive branch predictors use two major data structures, the *branch history register* (BHR) and the *pattern history table* (PHT). The branch history register is a $k$-bit shift register that records the execution history pattern of the most recent $k$ branches. Each time a branch completes, its outcome is shifted left into the register (1 for a taken branch, 0 for a not-taken branch). The $k$-bit branch execution pattern stored in the branch history register is used to select one of the $2^k$ entries in the pattern history table. Each of these entries contains a bimodal predictor to make a prediction based on the execution history of the specific pattern. There are three variations of branch history register implementations:

1. *G, for global*. A single global BHR records the history of the last $k$ branches encountered. This allows the branch predictor to take advantage of logical dependencies (also called *branch correlation*) between different branch instructions.

2. *P, for per-address*. There is a BHR for each branch and each BHR records the history of the last $k$ occurrences of the same branch. Therefore, the prediction only depends on the execution history of the branch itself and is independent of other branches. In practice, however, the number of
branches is different for each program and hardware resources are limited such that \( j \) bits of the branch address are used to index the \( 2^j \) branch history registers. The collection of multiple branch history registers is called \textit{branch history table}.

3. \( S \), for \textit{per-set}. This configuration is similar to the per-address configuration but branch instructions are grouped into sets. The branch history table can be implemented as a direct-mapped or set-associative cache.

Accordingly, there can be a global PHT (g), a PHT per branch address (p), or a PHT per set (s). This results in a total of nine different configurations of two-level adaptive branch predictors (Yeh and Patt, 1993). Figure 2.4 shows the simplest case of a two-level adaptive predictor, called GAg(k)\(^5\), which uses a single global \( k \)-bit BHR and a single global PHT.

![Figure 2.4: Implementation of a two-level adaptive predictor (GAg)](image)

Figure 2.5 on the next page shows an implementation of a PAp(k) two-level adaptive predictor that combines multiple per-address BHRs with multiple per-address PHTs. The first level of this predictor records the history of the last \( k \) occurrences of the same branch instruction, i.e. it uses self-history only. In this predictor configuration, each branch instruction has its own BHR and its own

\(^5\)The \( A \) in GAg stands for \textit{adaptive}. 

54
2.2. OVERVIEW OF BRANCH PREDICTION TECHNIQUES

PHT. However, as the number of branch instructions in a program may vary this causes implementation problems in practice and therefore the PAp predictor is mainly of theoretical interest (Šilc et al., 1999).

![Diagram of a local-history two-level adaptive predictor (PAp)](image)

Figure 2.5: A local-history two-level adaptive predictor (PAp)

Pan et al. (1992) propose a correlation-based prediction scheme that is similar to the global two-level predictors developed by Yeh and Patt (1991). The \((m,n)\)-correlation-based predictor proposed by Pan et al. (1992) uses the execution history of the most recent \(m\) branches to choose from \(2^m\) pattern history tables\(^6\) and selects an entry in the PHT using the lower \(k\) bits of the branch address. The update of the BHR works according to that used in two-level adaptive predictors. Alternatively, the \(2^m\) PHTs can also be implemented as a single \(2^{k+m}\) PHT. In this case, \(k\) bits of the branch address are concatenated with \(m\) bits of branch history to obtain the \(k+m\) index bits for accessing the PHT. McFarling (1993) calls this scheme \textit{gselect} and proposes to use a modified index function, which uses the bit-wise exclusive OR of the global branch history and part of the branch address to access the PHT. This predictor, which is called \textit{gshare}, exhibits a more random usage pattern in the PHT than a standard GAg predictor and makes interference between different branches less likely to occur. Branches that

\(^6\)Pan et al. (1992) use a different terminology. They use the terms \textit{branch prediction table} instead of PHT, and \textit{m-bit shift register} instead of BHR.
share the same execution history pattern now use different entries in the PHT because of their different addresses.

**Combined and Hybrid Branch Predictors**

Usually, some branches are best predicted using global history and branch correlation, while others are best predicted using local history only. Also, some predictors work best for floating-point intensive programs, while other predictors take advantage of the more complex control flow usually present in integer-dominated programs.

In addition to the gshare predictor, McFarling (1993) suggests combining the advantages of different branch predictors into a single predictor in order to achieve a better average misprediction rate than for each individual predictor. A *combined predictor* combines multiple predictors with a mechanism to select which predictor to use. The selection mechanism can either be static, i.e. the compiler decides at compile-time which predictor to use for each branch, or dynamic, i.e. the decision is made by a hardware mechanism during program execution. McFarling (1993) combines a gshare two-level adaptive predictor with a bimodal predictor and uses a two-bit bimodal predictor as selection mechanism. The introduction of combined predictors stems from the observation that different characteristics of branches cannot be optimally predicted by a single predictor. In principle, any branch prediction scheme can be included in a combined predictor and numerous variations have been presented and evaluated in research literature.

The combined branch predictor implemented in the Alpha 21264 microprocessor (Kessler, 1999) dynamically chooses between a local history and a global history two-level adaptive predictor to predict the outcome of a given branch instruction. The local history predictor (PAg) uses ten bits of the address of the current branch to index a 1024-entry table (BHT) in which each entry holds
ten bits of branch pattern history. The selected ten bits are used to index to a 1024-entry PHT of three-bit saturating counters. The global history predictor (GAg) records the outcome of the most recent 12 branches in a 12-bit BHR to index to a 4096-entry PHT of two-bit saturating counters. This 12-bit global history pattern is also used by the choice predictor to index to a 4096-entry table of two-bit predictors, which choose between the predictions made by the local and global predictors. Although the implementation of the dynamic branch prediction scheme in the Alpha 21264 microprocessors appears to be quite complex, the necessary resources required for its implementation (about 29k bits) are in fact small when compared to that of other units, for example, instruction and data caches.

A hybrid branch predictor\footnote{The terms combined branch predictor and hybrid branch predictor are often used interchangeably in literature.} proposed by Young and Smith (1994) is a modification of McFarling’s original combined branch predictor scheme (McFarling, 1993) and combines a compiler-based static branch predictor with a dynamic branch predictor. The PowerPC 620 is an example of a microprocessor using a hybrid branch predictor.

2.3 Low-Level Timing Analysis Techniques

The aim of low-level timing analysis is to model different micro-architectural features such that accurate bounds on WCET of basic blocks or individual instructions can be obtained. Low-level timing analysis can be divided into the following two areas (Engblom et al., 1999):

- **Local-effects analysis** estimates the WCET of individual instructions or sequences of instructions taking into account instruction pipelining itself and
the local effects of caching and branch prediction on pipelining (such as pipeline stalls caused by cache misses or stall cycles due to branch mispredictions).

- Global-effects analysis addresses the behaviour of caches and branch predictions within the whole program context. This analysis also takes into account the effects of context switches between tasks.

Obtaining tight WCET estimates for modern microprocessors requires that all hardware features are equally considered. Kim et al. (1999) present quantitative analysis results on how various sources of overestimation, such as infeasible paths, effects across basic blocks, instruction pipelining, instruction caching, and data caching, impact the accuracy of worst-case timing analysis. The impact of dynamic branch prediction on execution time analysis is not analysed.

### 2.3.1 Instruction Pipeline Modelling

In order to overcome the problem of excessive WCET estimates due to ignoring timing effects of instruction pipelines and caches, Lim et al. (1995) and Hur et al. (1995) describe a technique, called the extended timing schema (ETS), which extends the original timing schema proposed by Shaw (1989) such that timing effects of both pipelined execution and cache memory are taken into account. Each timing construct is associated with a worst-case timing abstraction (WCTA), which provides additional timing information about the surrounding program context. In order to limit the unpredictability of cache memory timing behaviour this approach assumes that cache partitioning is used to avoid bursts of cache misses due to task pre-emption. The extended timing schema approach determines the timing behaviour of basic blocks in terms of reservation tables. The rows of a reservation table represent the individual stages of the instruction pipeline and the columns represent clock cycles. Reservation tables show the
2.3. LOW-LEVEL TIMING ANALYSIS TECHNIQUES

allocation of pipeline stages for each clock cycle. Figure 2.6 shows an example of a reservation table for a simple sequence of assembler instructions. The number of columns in the reservation table represents the WCET of the instruction sequence.

(a) lw r1,(r4)
(b) slt r1,r31,r1
(c) beqz r1,loop
(d) addi r4,r4,#4

(a) DLX Code Example
(b) Reservation Table

Figure 2.6: Example of a reservation table

In the mid-1990s, microprocessors capable of issuing multiple instructions per clock cycle were announced by virtually every major microprocessor manufacturer. Examples of multiple-issue microprocessors that follow the superscalar principle include the Intel P5 (Pentium) and P6 (Pentium Pro, Pentium II, and Pentium III) families, the AMD-K5, K6, and K7 families, or the PowerPC 604/620, to name only a few. Most of the existing timing analysis techniques available at that time did not address the execution time behaviour of multiple-issue pipelines in their analysis.

In a more recent work, Lim et al. (1998) propose a worst-case timing analysis technique that, like their previous technique, is based on the extended timing schema. However, this technique now allows timing analysis for microprocessors with multiple instruction issue. Lim et al. assume that instructions are issued in program order (in-order issue). The microprocessor model uses the concept of delayed branches in order to reduce the pipeline penalties of taken branches. Basic blocks of such instruction set architectures always include the instruction following a conditional branch.

A directed acyclic graph, called instruction dependence graph (IDG), is con-
CHAPTER 2. LITERATURE REVIEW AND BACKGROUND

structured for each program construct. The nodes of the IDG represent instructions or sequences of instructions and the edges represent dependencies among instructions. Dependencies among instructions (data hazards) and limited processor resources (structural hazards) reduce both the possible degree of instruction overlap and the number of instructions that can be issued simultaneously. The weight of an edge (distance bound between instructions) is the minimum number of clock cycles required to resolve a dependency represented by the edge. Although this approach can accurately predict program execution time for multiple-issue microprocessors with in-order instruction issue, more advanced features, such as dynamic branch prediction and speculative execution, are not addressed.

Retargetability

A disadvantage of many low-level timing analysis techniques is that they are specific to a certain microprocessor architecture. Thus, adopting such techniques to different architectures in order to follow the pace of new microprocessor technologies is almost impossible. Harmon et al. (1994) describe a retargetable technique, called micro-analysis technique, to overcome this problem. This technique translates each assembler instruction into a sequence of micro-instructions using a set of translation rules. These micro-instructions allow the technique to account for caching and instruction overlap. However, the definition of micro-instructions requires detailed information about the micro-architecture of the target processor, which is difficult to obtain for proprietary microprocessors. Narasimhan and Nilsen (1994) present a tool, called the pipeline simulator compiler (psc), for portable execution time analysis. This tool uses an architecture description file that defines a model of the target processor to generate a simulation program for the target processor’s pipeline. Nilsen and Rygg (1995) complement the psc-generated pipeline simulator by another tool, called C path finder (cpf), which creates labels for basic blocks of C source code and determines a sequence of
labels representing the worst-case execution path of a program.

\begin{tabular}{|c|c|}
  \hline
  Block(s) & Execution Time \\
  \hline
  A & \( t_A = 9 \) \\
  B & \( t_B = 16 \) \\
  A \rightarrow B & \( t_{AB} = 20 \) \\
  \hline
\end{tabular}

(a) Simulation Runs

\begin{itemize}
  \item \( A \rightarrow B \)
  \item \( \delta_{AB} = -5 \)
  \item \( B \rightarrow t_B = 16 \)
\end{itemize}

(b) Timing Graph

Figure 2.7: Execution time effects of pipeline overlap

Engblom and Ermedahl (1999) take a different approach to address the re-targetability problem by using a trace-driven microprocessor simulator instead of pipeline modelling to determine the execution time effect of pipeline overlap between successive instructions. The calculation of timing effects is illustrated in Figure 2.7. Engblom and Ermedahl assume a microprocessor architecture that issues at most one instruction each cycle (single-issue or also called scalar instruction issue) and instructions execute in program order (in-order execution).

**Timing Anomalies**

Restricting the instruction pipeline to in-order issue avoids taking into account the timing anomalies reported by Lundqvist and Stenström (1999b) of superscalar microprocessors exploiting dynamic instruction scheduling and out-of-order execution. A timing anomaly is a situation where, for example, a cache hit actually causes the execution time of a sequence of instructions to be longer than for a cache miss. Dynamic scheduling schemes issue an instruction to an execution station as soon as all input operands are available for this instruction. The scheduling policy does not take into account the number of clock cycles required to execute each instruction and, therefore, the order of instructions may not always be optimal in terms of overall execution time. The presence of timing anomalies contradicts the common approach of traditional WCET analysis methods that
assume worst-case behaviour of instruction pipeline and cache always produces safe execution time estimates.

Lundqvist and Stenström present two methods for eliminating timing anomalies. The first method, called serial-execution method, assumes that all instructions execute sequentially without overlapping between instructions. For a superscalar microprocessor with an $n$-stage pipeline and an issue rate of $k$ instructions per clock cycle the estimated execution time can be as high as $nk$-times the theoretical execution time for pipelined execution. Of course, this method produces far too pessimistic execution time estimates.

The second method modifies the program code such that timing anomalies can no longer occur. Lundqvist and Stenström show that this method can provide less pessimistic but still safe WCET estimates compared to the serial-execution method.

### 2.3.2 Instruction Cache Analysis

Cache memory reduces the average latency of memory accesses and hence widens the bottleneck that stems from the increasing gap between microprocessor clock rate and main memory access time. Cache memory benefits from the principle of locality. Two orthogonal types of locality exist (Hennessy and Patterson, 2006):

- **Temporal locality.** A data item, or more generally, a memory location that is accessed recently is likely to be accessed again soon.

- **Spatial locality.** If a data item is accessed, it is likely that nearby items are accessed too. Programs typically show a high degree of spatial locality because instructions are accessed sequentially. This advantage is limited by changes of program flow, i.e. jumps, branches, or context switches between competing tasks scheduled by the operating system.
Despite the reduction of average memory access time they exhibit, many designers of real-time systems avoid using cache memories because of their unpredictable timing behaviour. This is due to the fact that cache memory behaviour is predictable rather than deterministic and thus complicates the estimation of tight WCET bounds. Many early static WCET analysis techniques did not address caching issues and assumed worst-case memory access times or completely ignored the effects of caches on WCET analysis.

Static Cache Simulation

Müller (1994) and Arnold et al. (1994) present a method called Static Cache Simulation that supports the provision of tight WCET estimates for direct-mapped instruction caches. Their method statically analyses the control flow of a program to predict the caching behaviour of instruction cache references. Each instruction is assigned a category according to its worst-case caching behaviour:

- **Always miss**: the instruction is *not guaranteed* to be in the cache when it is referenced.

- **Always hit**: the instruction is *guaranteed* to always be in the cache when it is referenced.

- **First miss**: the instruction is *not guaranteed* to be in the cache on its first reference each time the loop is executed, but is *guaranteed* to be in cache on subsequent references.

- **First hit**: the instruction is *guaranteed* to be in the cache on its first reference each time the loop is executed, but is *not guaranteed* to be in cache on subsequent references.

Healy et al. (1999) extend the Static Cache Simulator (Müller, 1994; Arnold et al., 1994) in order to bound the WCET effects for large code segments in the presence
of caches and a simple instruction pipeline based on the MicroSPARC implementation. Like in the original Static Cache Simulator, the control flow of a program is analysed to statically categorise the caching behaviour of each instruction as cache hits or misses. This information is then used by the pipeline path analysis to estimate the WCET of a sequence of instructions. Their experimental results show that tight WCET predictions can be produced but the evaluation only considers small sample programs and takes only into account a small direct-mapped cache. The pipeline model is limited in its complexity as it needs to be described by resource usage patterns of instructions.

**Associative Caches**

In contrast to direct-mapped caches, where only one cache block is subject to replacement in case of a cache miss, an $n$-way set-associative cache has $n$ blocks to choose from on a cache miss. There exist three primary replacement policies for selecting which cache block should be replaced:

- **Random**: the cache block to be replaced is selected randomly.

- **Least-recently used (LRU)**: the cache block that was referenced longest ago is replaced from the cache.

- **First-in, first-out (FIFO)**: replaces the cache block that was filled into the cache longest ago. This replacement policy is also called *round-robin*.

In a more recent work, Müller (2000) extends his previous work and presents a framework to provide upper bounds on the worst-case instruction cache performance for caches with arbitrary levels of associativity. The method implements an LRU replacement policy but he claims that it can be easily modified to use FIFO instead.
2.3.3 Data Cache Analysis and Virtual-Address Caches

Estimation of tight WCET is much more difficult for data caches than for instruction caches. While instruction addresses remain static during the execution of a program this is not necessarily the case for memory addresses of data references. Addresses of data references may even be unknown at compile-time and may change during program execution because, for example, addresses on the heap are calculated dynamically. This imposes a problem on the static execution time analysis of data caches.

Only a few methods in the area of WCET analysis for data caches have emerged over the past years. Rawat (1993) and later Basumallick and Nilsen (1994) use a graph colouring\textsuperscript{8} based technique to estimate the number of cache misses. For each memory reference (variable) in a program a live range is computed and cache lines are allocated to memory references such that different references do not interfere with one another for the same cache line. Therefore, access to memory only occurs on entry or exit of a live range. This approach guarantees that all references within a live range will hit the cache. A drawback of this technique is that no function calls and dynamic memory references are supported.

White et al. (1999) extend their existing WCET analysis framework (see previous section) with an approach to determine upper bounds on the worst-case performance of data and wrap-around caches (for example, as used in the MIPS R10000). Their approach determines a bounded range of relative addresses for each data reference from the data-flow information produced by the vpo compiler. The relative address ranges are then translated into virtual address ranges by examining the order of data declarations and the call graph of the program. The

\textsuperscript{8}The graph colouring problem is to colour the nodes of a graph with \( n \) colours such that no adjacent nodes have the same colour.
static cache simulator categorises the data references and a timing analyser uses these categorisations to estimate the worst-case cache behaviour for each loop and function in a program. Like other data cache analysis methods, the static cache simulator cannot calculate addresses in the heap but only static data, which consists of global variables, static variables, and non-scalar constants. Their implementation restricts the analysis to direct-mapped caches. Although function calls are supported, they are not allowed to be recursive nor indirect.

Dynamic data references to memory seem to remain a dominant problem for static analysis of data cache behaviour. Lundqvist and Stenström (1999a) tackle this problem from a different perspective. They analyse what ratio of static versus dynamic memory references can be expected for the SPEC95 benchmark suite and find that in fact more than 84% of the data references are predictable. Their method is based on the notion of distinguishing between predictable and unpredictable data references. Unpredictable memory references are mapped to data structures either by the compiler or by the programmer. A data structure is marked as unpredictable if there is at least one unpredictable memory reference mapped to it. All unpredictable data structures are excluded from caching, for example, by mapping them into a dedicated un-cached memory area. This approach is different from previously published work on data cache analysis, which assigns a miss penalty of two cycles to each unpredictable cache access. Although Lundqvist and Stenström (1999a) do not explicitly consider the effects of preemptive scheduling policies they propose to use cache partitioning to assign disjoint cache areas to different processes. Furthermore, cache partitioning can also be used to map predictable and unpredictable data structures to different cache partitions in order to allow caching also for unpredictable references. They conclude that their approach allows them to tighten estimation of worst-case data cache performance for unpredictable data structures.

Ghosh et al. (1997) propose the Cache Miss Equations (CME) framework
2.3. LOW-LEVEL TIMING ANALYSIS TECHNIQUES

in order to statically model data cache behaviour in loop-oriented code. Their method characterises the cache behaviour by a set of linear Diophantine equations. However, solving these equations is generally considered impractical. Probabilistic methods for reducing the computational complexity have been proposed but they introduce some pessimism when estimating the number of cache misses. Also, they impose restrictions on the code, for example, by assuming perfectly nested, rectangular loops with no data dependent conditionals.

Based on the original CME framework, Ramaprasad and Müller (2005) propose an approach to extend the analysis to more arbitrary programs. This is achieved by a transformation called forced loop fusion to transform any arbitrary loop nest into a single loop nest, as required by the original CME framework developed by Ghosh et al. (1997). Ramaprasad and Müller report improvements on the accuracy of worst-case data cache behaviour of up to two orders of magnitude over the original analysis framework.

Virtual-to-Physical Address Translation

In computer systems using virtual memory, the virtual address generated by the ALU of the microprocessor is dynamically mapped onto the physical memory of the system during run-time. The translation of virtual addresses to physical addresses may require additional clock cycles and, thus, may further delay cache access. Most microprocessors store recent address translations in a special cache called translation look-aside buffer (TLB) to speed-up the translation process. As with data caches, a TLB miss caused by a data access may generate or contribute to a timing anomaly. Nevertheless, transaction look-aside buffers are ignored by most static analysis techniques.

A physical cache is accessed by the physical address only and, therefore, virtual-to-physical address translation is required prior to accessing the cache.
Both Müller (2000) and White et al. (1999) use a physical cache for their analysis and assume that the system page size is an integer multiple of the cache size. By using this assumption, virtual addresses can be used to predict the behaviour of the physical cache.

Alternatively, some recent microprocessors use virtual addresses to access the first-level cache, which is then called a virtual cache. This avoids the delay for address translation as long as the cache hits. Also, the above assumption for static analysis of physical caches is no longer necessary. Yet, the potential impact of virtual-to-physical address translation on execution time has to be considered, and there has only been little work reported on this issue to date. Bennett and Audsley (2001) look at flexible predictable virtual addressing as a means of providing spatial separation for processes with different integrity levels in safety-critical systems.

### 2.3.4 Analysis of Dynamic Branch Prediction Techniques

Branch prediction is a widely-used technique for overcoming the problem of microprocessor pipeline stalls caused by control dependencies. Such dependencies arise from conditional branches and are a major limiting factor to microprocessor performance because the outcome of branches must be resolved before new instructions can be fetched from memory. The Compaq Alpha 21264 microprocessor, for example, takes at least seven clock cycles to resolve a branch (Kessler, 1999). This problem is particularly critical for microprocessors that attempt to issue multiple instructions per clock cycle. Dynamic branch prediction solves this problem by predicting the outcome of branches based on the history of previous executions. This allows the microprocessor to continue to fetch new instructions from the predicted path and execute these instructions speculatively. Of course, misprediction can occur, thus requiring the unit to abort the speculated instructions. Sophisticated branch prediction techniques, however, achieve aver-
age prediction rates of over 90% (Yeh and Patt, 1991). Static timing analysis for real-time systems, on the other hand, is more interested in obtaining accurate upper bounds on worst-case prediction rates rather than average case performance.

**Prediction of Conditional Branches**

The first work on execution time analysis for microprocessors using dynamic branch prediction techniques is presented by Colin and Puaut (1999). Their method is based on static program analysis and models the *branch target buffer* (BTB) of an Intel Pentium microprocessor in order to statically provide an upper bound on the number of timing penalties caused by branch mispredictions. A branch target buffer, often called *branch target address cache* (Hennessy and Patterson, 2006), is a very simple dynamic branch prediction technique used by some of the earlier superscalar microprocessors. A BTB consists of a table with branch addresses, the corresponding branch target addresses, and prediction information (Lee and Smith, 1984). The BTB of the Pentium processor (P5 family) is implemented as a four-way set associative cache with 256 entries using a least-recently-used (LRU) replacement policy. If a branch instruction is not present in the BTB the branch is predicted to be *not-taken* by default. The Pentium uses two state bits for each entry in the BTB to implement a two-bit saturation up-down counter.

The approach is to categorise branch instructions so that their worst-case impact on program execution time can be estimated. Colin and Puaut apply the idea of the classification scheme used in the static cache simulator (Müller, 1994) to worst-case analysis of branch prediction and define the following four categories:

- **Always default-predicted**: the branch instruction is always predicted *not-taken* (default-predicted).
• **First default-predicted**: the branch instruction is predicted *not-taken* for its first execution. Subsequent executions are predicted according to the branch history recorded in the BTB (history-predicted).

• **First unknown**: the prediction state for the first execution of the branch instruction is *unknown* but it is predicted according to its history for subsequent executions.

• **Always unknown**: the prediction state for the branch instruction is *always unknown*.

The purpose of this categorisation is to determine for each branch instruction whether or not it is present in the BTB at a particular execution instance. The subset of branch predictions that are guaranteed to be correct can be determined by using this information. Colin and Puaut assume that history-predicted branches are always predicted correctly, except for the exit condition of loops, which is always predicted incorrectly. For conditional constructs as shown in Figure 2.8(a) on the facing page the path with the highest execution time is regarded as an unconditional jump for further analysis, i.e. predictions based on the history in the BTB are always correct, if the following condition holds true:

\[ T(S_1) > T(S_2) + \delta \quad \vee \quad T(S_2) > T(S_1) + \delta, \quad \delta > 0, \quad (2.5) \]

where \( \delta \) is the worst-case timing penalty due to a branch misprediction.

Otherwise, it is assumed that the branch is always mispredicted (pessimistic assumption) and the instruction is categorised as *always unknown*.

Now, let us assume that one of these two paths has a higher execution time than the other, for example, \( T(S_1) > T(S_2) \). Furthermore, we assume that the conditional construct shown in Figure 2.8(a) on the next page is repeated several times, for example, because it is located within a surrounding loop as indicated.
by the dotted back-edge in the control flow graph depicted in Figure 2.8(b). This repeated execution allows the bimodal branch predictor to assume different states. In this case, the worst-case scenario for a two-bit branch predictor is a branch instruction alternating between the *taken* and *not-taken* path.

\[
\begin{align*}
\text{if } B \text{ then} & \quad S_1 \\
\text{else} & \quad S_2 \\
\text{end if}
\end{align*}
\]

(a) Source Code  
(b) Control Flow Graph

Figure 2.8: Conditional statement example

According to Colin and Puaut, the worst-case branch predictor scenario occurs when the branch alternates to the $S_2$ path after the execution of the $S_1$ path and if the condition $T(S_2) > T(S_1) + \delta$ is true. Then, the next execution of $S_2$ would result in a correct branch prediction. Since $T(S_1) + \delta > T(S_2)$ is trivially true, the branch instruction changes its direction after the execution of $S_2$ again and causes another misprediction. After this, the scenario repeats itself and produces an alternating sequence of *taken* and *not-taken* paths. The condition provided by Colin and Puaut does not take into account the additional misprediction penalty caused by the second state transition and therefore is not complete. Instead, the following condition applies, as briefly discussed above and shown in Bate and Reutemann (2004):

\[
T(S_1) > T(S_2) + 2\delta \quad \lor \quad T(S_2) > T(S_1) + 2\delta. \tag{2.6}
\]

Although this condition appears to be very promising in reducing the number of branches with assumed worst-case behaviour, it remains to be evaluated how
frequently this condition actually occurs in practice. This question is linked to
the relevance criterion defined earlier in Section 1.5 and will be assessed through
the evaluation framework in the next chapter.

Colin and Puaut conclude that the behaviour of a BTB is predictable enough
to allow for a tight estimate of program WCET. Recent microprocessors use more
complex branch prediction techniques, for example, two-level adaptive or even
combined branch predictors. It is questionable whether the approach presented
by Colin and Puaut can be easily adapted to take into account these techniques.
Nevertheless, the earlier work published by Colin and Puaut (1999) triggered the
development of a number of new static WCET analysis techniques for modelling
the execution time effects of branch predictors.

Engblom (2002, 2003) analyses the execution time variance caused by branch
prediction mechanisms for the Intel Pentium III (Intel, 1999), AMD Athlon
(AMD, 2002), and UltraSPARC III (Horel and Lauterbach, 1999) microprocessors. Engblom evaluates the execution times of a nested loop construct for dif-
ferent numbers of iterations of the inner loop while the number of iterations of
the outer loop remains fixed. He observes that the execution time is sometimes
lower for increasing number of loop iterations, which is inverse to the expected
behaviour. This effect is called inversions.

Using the results for the AMD Athlon microprocessor as an example, the
total execution time rises significantly between the ninth and tenth iteration of the
inner loop. The conclusion of his work is that complex dynamic branch prediction
schemes should not be used in real-time systems. Unfortunately, Engblom does
not provide an explanation as to why these effects occur.

Rochange and Sainrat (2002) discuss potential effects of speculative execution
in the context of static WCET analysis. They conclude that traditional static
WCET analysis techniques cannot accurately model the effects of speculatively
executing instructions along a mispredicted path. This is because it requires a
complex interaction between the high-level (program flow) and low-level timing analysis, thus limiting the amount of modularity among the various analysis models.

Bodin and Puaut (2005) propose a WCET-oriented static branch prediction scheme for the PowerPC 7451 microprocessor architecture (Freescale, 2005), which supports both static and dynamic branch prediction. The branch prediction unit can be configured to use static branch prediction instead of dynamic prediction for all branch instructions by a dedicated processor register. In order to minimise the WCET estimates of a program, Bodin and Puaut implement an iterative algorithm to select the static prediction direction on a per-branch basis. They claim that this algorithm could be integrated rather easily in the back-end of a compiler, however, without providing further details. The experimental results on the selected microprocessor architecture show a reduction by up to 21% of the WCET estimates compared to the case where all branch instructions are considered mispredicted. Although these results are quite promising, they do not compare their results with the ones achieved by static analysis techniques for dynamic branch predictors.

Prediction of Indirect Branches

Indirect branches transfer control to a target address stored in a register. The target address or the offset relative to the program counter are no longer encoded in the instruction and therefore it is not known at compile-time. Although indirect branches are not as frequent as direct branches, indirect branches are often used in object-oriented programs to implement late binding of method invocations, i.e. polymorphic calls. Current microprocessors use a branch target buffer (BTB) to predict indirect branches. However, the misprediction rate for indirect branches, typically about 25%, is significantly higher than for target address prediction of conditional branches. Although there has been no work reported on
analysing the impact of indirect branches on WCET analysis, various researchers consider possible predictions schemes to improve the accuracy of indirect branch prediction.

Driesen and Hölzle (1997), for example, investigate the predictability of indirect branches to determine what factors cause the inferior accuracy of indirect branch prediction compared to prediction of conditional branches. They find that the small number of indirect branches found in the SPECint95 benchmark programs compared to typical C++ programs indicate that the SPEC benchmark suite is not appropriate for evaluating indirect branch prediction mechanisms. Their evaluation results indicate that indirect branch prediction using branch target buffers can be significantly improved by applying the concept of two-level and hybrid predictors to the target address prediction of indirect branch instructions.

Techniques for the target address prediction of indirect branches and their possible impact on static WCET analysis will not be considered in further detail in this thesis. Indirect branches associated with `switch` statements will be discussed in Section 4.4.3 on page 145.

2.4 Summary

Several techniques for obtaining WCET figures or estimates of real-time programs have emerged over the past two decades since the first publications on static WCET analysis were presented by Shaw (1989) and Puschner and Koza (1989). Since then, WCET analysis has become an active area of research in the domain of real-time systems and a number of analysis approaches have been developed by various research groups (Puschner and Burns, 2000).

The implementation of branch target buffers or branch target address caches is quite similar to the design of an instruction cache. This leads to the assump-
tion of some researchers that existing techniques for static analysis of instruction caches can be extended to address dynamic branch prediction (Colin and Puaut, 1999). Today, however, most microprocessors use more sophisticated branch prediction techniques (e.g. two-level or hybrid predictors) for the prediction of branch direction and branch target address caches are only used for the prediction of branch target addresses if the branch direction is predicted \textit{taken}. For example, branch prediction techniques that use global history take advantage of the correlation between branch instructions. The impact of such techniques on worst-case execution analysis has received less attention in the academic community than other areas of low-level WCET analysis techniques, such as modelling instruction pipelines or caches.

Engblom (2002, 2003) analyses the execution time variance caused by branch prediction mechanisms for the Intel Pentium III, AMD Athlon, and UltraSPARC III microprocessors when executing nested loop constructs. Based on his observations, which are sometimes inverse to the expected behaviour, he concludes that complex dynamic branch prediction schemes should not be used in real-time systems. Engblom did not provide an explanation as to why these execution time variations occur. A theoretical model of branch predictor behaviour should provide the missing explanation for bimodal and global history branch predictors.

While caches and dynamic branch prediction techniques have similarities, some differences make it difficult if not impossible to extend existing analysis techniques for caches to support analysis of dynamic branch predictors. Nevertheless, it seems that there exist some ideas that can be applied to branch prediction analysis as well. In the context of static analysis of data caches, Lundqvist and Stenström (1999a) distinguish between predictable and unpredictable data references. All unpredictable data structures are excluded from caching, for example, by mapping them into a dedicated un-cached memory area. Colin and Puaut (1999) apply the idea of the classification scheme used in the static cache
simulator (Müller, 1994) to worst-case analysis of branch prediction.

In summary, the survey of previously published work in the area of low-level timing analysis techniques revealed that existing approaches, for example by Colin and Puaut (1999), for modelling dynamic branch predictor behaviour should be augmented along the following three main aspects:

- Establish a theoretical model of branch prediction behaviour with special focus on worst-case predictor states in order to define worst-case execution pattern for branches. The pessimism introduced by the analysis model should be less than the overall performance benefit gained by using a particular branch prediction technique. Basically, this defines the required level of accuracy of the estimated WCET figures;

- Integrate the analysis approach into an overall analysis framework to demonstrate its modularity; and finally

- Demonstrate the relevance of the approach for a large number of branches in a program by distinguishing between branches that are predictable and not predictable statically during compile time.
Chapter 3

Empirical Evaluation of Branch Predictors

This chapter describes the simulation methodology and metrics used to establish an evaluation framework for branch prediction analysis. The evaluation framework analyses the branch prediction behaviour of several sample programs in order to derive a classification scheme for branch instructions from the simulation results.

In this chapter, Section 3.1 introduces the basic terminology and metrics used by the evaluation framework. Section 3.2 describes the simulation environment that is used to run the simulations of the sample programs. The environment is based on the SimpleScalar simulator suite, which is presented in more detail. Also, the branch predictor configurations that are used for the simulations are presented. The characteristics of the selected sample programs and general simulation results related to branch predictor performance are presented in Section 3.3 and distributions of branch mispredictions for each sample program are discussed in Section 3.4. Section 3.5 presents two branch classification schemes that classify branches according to their dynamic execution behaviour. Section 3.6 illustrates the effects of branch prediction on the performance of the instruction pipeline.
CHAPTER 3. EMPIRICAL EVALUATION OF BRANCH PREDICTORS

using a cycle-level simulator. Interference between branches limits the accuracy of branch prediction and complicates the analysis of individual branches because the predictor state depends on the behaviour of different branch instructions. Finally, Section 3.7 summarises the results of this chapter.

3.1 Introduction

The primary goal of the classification schemes that are presented later in this chapter (see Section 3.5 on page 94) is the partitioning of branches into a set of branch instructions whose dynamic behaviour can be determined statically and into a set of branch instructions that exhibit complex or non-deterministic execution patterns.

As already discussed in the previous chapter, this problem is similar to static analysis for data caches. In a similar fashion to memory references for load/store instructions, branch conditions depend on input data that is not always known at compile-time. If the branch outcome cannot be predicted statically we must assume the worst-case behaviour in terms of branch prediction, i.e. each branch instance will cause a stall of the instruction fetch engine due to a branch misprediction. Lundqvist and Stenström (1999a) observe that more than 84% of the data accesses are in fact predictable, using sample programs from the SPEC95 benchmark suite. As far as branch prediction is concerned, we will show in this chapter that, firstly, the behaviour of a large number of branch instructions is predictable at compile-time and that, secondly, the majority of branch mispredictions are caused by only a small number of difficult-to-predict branches.

The primary metric used for measuring and comparing branch predictor performance is the branch misprediction rate, which is defined as the ratio between the number of dynamic branches that were mispredicted and the total number of dynamic branch instructions. In this context, two basic terms will be
used throughout this thesis: static instructions and dynamic instructions. These terms are defined by the following two definitions.

**Definition 3.1 (Static Instruction)**

Static instructions encompass all assembler instructions defined in the object code of a program. All static instructions can be uniquely identified by their instruction address.

**Definition 3.2 (Dynamic Instruction)**

Dynamic instructions are instances of static instructions created during the execution of a program.

The mapping between the set of static instructions and the set of dynamic instructions is surjective\(^1\) but not injective\(^2\). The sequence of dynamic instructions represents an execution trace for a particular program run.

The advantage of using the branch misprediction rate as a metric is that it is affected only by the run-time behaviour of the program under consideration and the configuration of the branch prediction mechanism itself. Using this metric instead of, for example, the number of clock cycles required or the execution time itself, allows us to examine and model branch predictor performance independently from other hardware design aspects, such as instruction pipelines or number of clock cycles per second.

### 3.2 Simulation Environment

The SimpleScalar microprocessor simulator tool set (Burger and Austin, 1997; Austin et al., 2002), which has been developed at the University of Wisconsin-

---

\(^1\)The mapping \(a: X \mapsto Y\) is called injective or one-to-one if for each \(y \in Y\) there exists at most one \(x \in X\) such that \(a(x) = y\).

\(^2\)The mapping \(a: X \mapsto Y\) is called surjective or onto if for each \(y \in Y\) there exists at least one \(x \in X\) such that \(a(x) = y\).
Madison, is used throughout this work for empirical evaluation of various branch predictor configurations. This simulator suite is well recognised in the academic community for the evaluation of different microprocessor architectures (Austin et al., 2002) and in textbooks on microprocessor architecture (Hennessy and Patterson, 2006). Before using it in the context of this thesis, a comprehensive evaluation of it was performed to check its suitability. In addition, the branch prediction results gathered from SimpleScalar have been confirmed by a separate Perl script, which has been used for the analysis of interference between branch instructions (see also Section 4.6 on page 158) by modelling branch predictor behaviour for an execution trace of branch instructions.

The SimpleScalar architecture is derived from the MIPS-IV instruction-set architecture (ISA). The 64-bit MIPS-IV instruction-set architecture (Price, 1995) was first defined in 1994 and is implemented in the R5000, R7000, R8000, and the R10000 (Yeager, 1996) microprocessors. The most notable difference compared to the MIPS-IV ISA is that the SimpleScalar ISA does not use delay slots for loads, stores, and control transfers. SimpleScalar does not resemble an existing microprocessor architecture but models an abstract machine with a five-stage instruction pipeline and a memory architecture supporting instruction and data caches. The centralised execution core of the simulator is based on the Register Update Unit (RUU) design (Sohi, 1990), which combines the physical register file, reorder buffer, and instruction issue window into a single structure.

Because of its abstract machine model, the SimpleScalar cycle-level simulator does usually not provide cycle accurate performance figures compared to real microprocessor architectures and tends to overestimate the performance as evaluated by Desikan et al. (2001) for the Alpha 21264 architecture (Kessler, 1999). Taking into account this limitation, we do not aim at modelling a particular microprocessor implementation in this thesis but simply use SimpleScalar as reference architecture for comparing different branch predictor configurations.
3.2. **SIMULATION ENVIRONMENT**

![Diagram of Simulation Environment](image)

Figure 3.1: Simulation environment
CHAPTER 3. EMPIRICAL EVALUATION OF BRANCH PREDICTORS

Figure 3.1 on the preceding page shows an overview of the simulation and evaluation framework used in this work. An AMD Athlon 1.2 GHz PC running SuSE Linux 7.3 is used as a host system. The evaluation of the branch prediction performance of a sample program is described in the following. The GNU C Cross Compiler (Version 2.6.3) translates the C source files into binary code for the SimpleScalar architecture. The generated binary code is compatible with the MIPS ECOFF object format. Basically, the simulators available in the SimpleScalar tool set provide two different levels of simulation accuracy:

- **instruction-level simulators** such as `sim-cache` for cache and `sim-bpred` for branch predictor evaluation; and a

- **cycle-level simulator**, called `sim-outorder`, providing a detailed model of an out-of-order instruction pipeline with caches and branch prediction.

The advantage of simulation at the instruction-level is that it has a shorter overall simulation time than that of a cycle-level simulator, but this comes at the price of less accuracy in terms of simulation depth. The simulation speed of the `sim-bp` simulator (including generation and compression of the branch trace file) is about 1.1 MIPS on an AMD Athlon PC running at 1.2 GHz, while `sim-outorder` achieves about 0.5 MIPS. Yet, this lower level of simulation depth is sufficient to evaluate the branch prediction behaviour of a program. Full instruction pipeline simulation, on the other hand, is required to evaluate the impact of branch prediction on overall microprocessor performance beyond the level of counting the number of branch mispredictions.

The extended scope of simulation provided by the `sim-outorder` simulator is the subject of Section 3.6 on page 100 where we compare the microprocessor performance figures for typical and worst-case predictor configurations and provide an assessment of the potential pessimism that is involved with static WCET analysis of branch predictors.
3.2. SIMULATION ENVIRONMENT

The \texttt{sim-bp} branch prediction simulator reads in the binary code of the sample program together with its input data. This simulator has been modified by the author to extend the original \texttt{sim-bpred} simulator in order to support the output of a detailed text-based branch instruction trace for each simulation run. The branch instruction trace describes for each instance of a branch the instruction address, the predictor state, and the predicted and actual target addresses. From this information, Perl scripts generate misprediction figures and perform a branch pattern classification for each branch instruction. This level of branch predictor performance analysis is not possible with the original set of SimpleScalar simulator tools.

Each of the sample programs, which we will discuss in more detail in the following section, is simulated with a set of different branch predictor configurations. These are defined in Table 3.1 on the following page. This selection of branch predictors covers the most widely used predictor types (Evers, 2000; Hennessy and Patterson, 2006): static predictors, bimodal predictors using local branch history, two-level branch predictors, and combined branch predictors. The PAg and PAp predictors are mainly of theoretical interest because of the hardware resources needed to implement these predictors in practice (Šilc et al., 1999). Nevertheless, they have been included in this evaluation framework in order to assess the performance of two-level branch predictors using local branch history.

As far as static analysis is concerned in the context of this thesis, the focus will be on the bimodal branch predictor (Chapter 4) and the GAg two-level global history predictor configuration (Chapter 6).

The \texttt{sim-bp} simulator provides a simple command-line notation for defining various static and dynamic branch predictor configurations. The trace file generated by the branch predictor simulator represents the sequence of dynamic branch instructions encountered during program execution. For each branch instruction in the trace the branch address, predicted and resolved direction, and
### Predictors and Configurations

<table>
<thead>
<tr>
<th>Predictor</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>bimod</td>
<td>4096-entry BHT</td>
</tr>
<tr>
<td>comb</td>
<td>4096-entry combining predictor with saturating up/down counters selects between a 4096-entry bimodal predictor and a GAg two-level predictor using a 12-bit BHR.</td>
</tr>
<tr>
<td>GAg</td>
<td>Two-level adaptive predictor using a single 12-bit BHR to index a global 4096-entry PHT.</td>
</tr>
<tr>
<td>GAp</td>
<td>Two-level adaptive predictor using a single 12-bit BHR to index four 4096-entry PHTs. The PHT is selected by the branch address.</td>
</tr>
<tr>
<td>gshare</td>
<td>GAg two-level predictor using a global 12-bit BHR xor-ing the history and branch instruction address to index a global 4096-entry PHT.</td>
</tr>
<tr>
<td>PAg</td>
<td>Two-level adaptive predictor using a set of 512 12-bit BHRs, selected by the branch address, to index a global 4096-entry PHT.</td>
</tr>
<tr>
<td>PAp</td>
<td>Two-level adaptive predictor using a set of 512 12-bit BHRs to index four 4096-entry PHTs. Both the BHR and the PHT are selected by the branch address.</td>
</tr>
<tr>
<td>taken</td>
<td>Static branch predictor that predicts all conditional branch instructions as being taken, i.e. instruction fetching always continues at the branch target address.</td>
</tr>
</tbody>
</table>

Table 3.1: Simulated branch predictor configurations
the predicted and resolved target address are recorded. The size of the compressed branch trace files for the four sample programs is in the range of 10 to 80 MBytes.

In addition to the branch trace the simulator generates a report with various performance figures, such as simulation time, total number of instructions, instruction class distribution, and mean branch misprediction rate. A Perl script extracts a table of static branch instructions from the binary code and analyses the branch trace file in order to collect branch profiling information for each static branch instruction. Finally, the branch profiling information is presented graphically using the \texttt{R} statistics program. A brief overview of this tool and the philosophy behind the \texttt{R} language is provided by Ripley (2001).

3.3 Sample Programs

The set of sample programs that have been chosen for the evaluation of branch predictor behaviour consists of four programs. Table 3.2 on the next page provides an overview of the sample programs. The \texttt{gcc} sample program is part of the SPECint95 integer benchmark suite, for which executables are provided together with the SimpleScalar tool set. This sample program represents the first stage (\texttt{cc1}) of the GNU C Compiler, i.e. it translates a pre-processed input file into an assembler source file. The \texttt{gcc-opt} sample program uses exactly the same executable as \texttt{gcc} but the program is executed with its option \texttt{-O2}, which enables optimisation of the generated code. Using this option increases the number of static branch instructions being executed by the sample program. In other words, the coverage of branch instructions executed at least once is higher for \texttt{gcc-opt} than for \texttt{gcc}. In our suite of sample programs, \texttt{gcc} represents programs with complex functional logic, a high diversity of branch instructions, and a strong dependence on the format and contents of the provided input data, which is a
pre-processed C source file in this case.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc-opt</td>
<td>GNU C Compiler version 2.6.3 from SPECint95 with option -O2.</td>
</tr>
<tr>
<td>gcc</td>
<td>GNU C Compiler version 2.6.3 from SPECint95 with no options.</td>
</tr>
<tr>
<td>bladeenc</td>
<td>Freeware MP3 encoder.</td>
</tr>
<tr>
<td>bzip2</td>
<td>Open-source high-quality data compressor.</td>
</tr>
</tbody>
</table>

Table 3.2: Description of the sample programs

Both the bladeenc and bzip2 have been selected as sample programs because a large number of instructions are executed within only a few loops. Therefore, these two sample programs provide a higher level of code locality compared to gcc, where the executed instructions are distributed over a wider portion of the object code. The bladeenc sample program uses a significant amount of floating-point processing and exhibits less complex functional logic.

<table>
<thead>
<tr>
<th>Program</th>
<th>Cond Branches</th>
<th>Uncond Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Used</td>
</tr>
<tr>
<td>gcc-opt</td>
<td>38,193</td>
<td>17,627</td>
</tr>
<tr>
<td>gcc</td>
<td>38,193</td>
<td>8,130</td>
</tr>
<tr>
<td>bladeenc</td>
<td>3,440</td>
<td>1,702</td>
</tr>
<tr>
<td>bzip2</td>
<td>2,877</td>
<td>1,058</td>
</tr>
</tbody>
</table>

Table 3.3: Static branch instruction distribution

Table 3.3 lists the distribution of static branch instructions for each sample program. The total number of conditional branches in each program is listed in the second column. The third and fourth column lists the number and ratio...
in percent of conditional branches that were executed at least once during the simulation run of the program. The percentage figure corresponds to the code coverage of unconditional branch instructions. Similarly, the second part of the table lists information about unconditional branches. It can be observed from Table 3.3 that the coverage of conditional and unconditional branch instruction is doubled for the gcc-opt sample program compared to gcc. Also, the total number of executed branch instructions for the gcc-opt sample program is significantly higher than for the remaining three sample programs.

Table 3.4 shows the total number of all instructions executed for each sample program. The last column in this table gives the average number of instructions per branch (IPB), which is the ratio between the total number of executed instructions and the total number of executed branches. The high number of instructions per branch for the bladeenc sample program is typical for most floating-point intensive programs (see also Table 3.5 on the following page, last column). In contrast, the gcc sample program is representative of an integer-oriented program with complex functional logic. The remaining columns show the breakdown of the two basic types of branch instructions: conditional branches and unconditional branches, which are also referred to as jumps.

<table>
<thead>
<tr>
<th>Program</th>
<th>Total Inst</th>
<th>Branch Inst</th>
<th>IPB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Cond</td>
</tr>
<tr>
<td>gcc-opt</td>
<td>1,950,453,119</td>
<td>388,059,771</td>
<td>300,025,370</td>
</tr>
<tr>
<td>gcc</td>
<td>408,205,969</td>
<td>84,571,244</td>
<td>67,133,373</td>
</tr>
<tr>
<td>bladeenc</td>
<td>812,864,143</td>
<td>109,526,481</td>
<td>75,280,109</td>
</tr>
<tr>
<td>bzip2</td>
<td>362,023,742</td>
<td>50,801,610</td>
<td>47,076,363</td>
</tr>
</tbody>
</table>

Table 3.4: Execution frequency of instructions

The simulation results show that among the two types of branch instructions, conditional branches are executed most frequently for the four sample programs.
Also, taking the ratio between the number of branches and the total number of instructions a branch can be expected every five to seven instructions on average. Unconditional branches can be further split into immediate jumps, returns, and indirect jumps. Both conditional and unconditional branch instructions need a mechanism that provides the target address already in an early stage of the instruction pipeline. This mechanism is referred to as address prediction and is typically achieved by a branch target buffer (BTB) or a branch target address cache (BTAC). The mechanisms for branch direction prediction and branch address prediction can be combined, e.g. single BTB, or separated, e.g. dynamic branch predictor and a BTB. However, for the remainder of this thesis we will concentrate on conditional branch instructions and do not consider the effects caused by address prediction. Within this work we simply write branch prediction when we mean branch direction prediction.

<table>
<thead>
<tr>
<th>Program</th>
<th>Load</th>
<th>Store</th>
<th>Jump</th>
<th>Branch</th>
<th>Int</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc-opt</td>
<td>26.8%</td>
<td>14.9%</td>
<td>4.5%</td>
<td>15.4%</td>
<td>38.4%</td>
<td>0.0%</td>
</tr>
<tr>
<td>gcc</td>
<td>23.3%</td>
<td>14.0%</td>
<td>4.3%</td>
<td>16.4%</td>
<td>42.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>bladeenc</td>
<td>22.3%</td>
<td>7.3%</td>
<td>4.2%</td>
<td>9.3%</td>
<td>38.3%</td>
<td>18.7%</td>
</tr>
<tr>
<td>bzip2</td>
<td>25.7%</td>
<td>13.1%</td>
<td>1.0%</td>
<td>13.0%</td>
<td>47.2%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Table 3.5: Dynamic distribution of instruction classes

Table 3.5 shows the distribution of different types of instructions that were executed during the simulation run of the four sample programs. The high number of load and store instructions is characteristic for a load/store architecture, which is a common feature of RISC microprocessors.

The columns in Table 3.6 on the next page detail the ratio of static branch instructions that are taken (T), have a backward (B) or forward (F) branch target address, are backward taken (BT) or forward not-taken (FN). Branches at the end of a loop are typically biased towards the backward taken direction.
3.4. DISTRIBUTIONS OF BRANCH MISPREDICTIONS

<table>
<thead>
<tr>
<th>Program</th>
<th>T</th>
<th>B</th>
<th>F</th>
<th>BT</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc-opt</td>
<td>50.5%</td>
<td>19.2%</td>
<td>80.8%</td>
<td>13.5%</td>
<td>43.8%</td>
</tr>
<tr>
<td>gcc</td>
<td>57.7%</td>
<td>22.0%</td>
<td>78.0%</td>
<td>15.9%</td>
<td>36.2%</td>
</tr>
<tr>
<td>bladeenc</td>
<td>72.8%</td>
<td>12.3%</td>
<td>87.7%</td>
<td>10.4%</td>
<td>25.3%</td>
</tr>
<tr>
<td>bzip2</td>
<td>74.1%</td>
<td>61.8%</td>
<td>38.2%</td>
<td>58.8%</td>
<td>22.9%</td>
</tr>
</tbody>
</table>

Table 3.6: Branch direction distributions

Forward branches are usually associated with the condition in if-statements and the table shows the percentage of such forward branches being not-taken. Simple static branch prediction schemes take advantage of such behaviour and the ratios shown in Table 3.6 give an idea of how accurate such predictors would be for the four sample programs. This will be discussed in more detail in the following paragraph.

3.4 Distributions of Branch Mispredictions

Mispredicted branches disrupt the smooth flow of instructions into the microprocessor because the pipeline has to be flushed once a misprediction is detected. This has a negative impact on the efficiency of the instruction fetch mechanism and, thus, ultimately limits the overall performance of the microprocessor because additional clock cycles are required to remove the instructions fetched from the wrongly speculated path.

The average\(^3\) misprediction rates for three configurations of static branch predictors are shown in Table 3.7 on the next page. The taken predictor simply predicts all branches to be taken, as defined earlier in Table 3.1 on page 84. The taken predictor performs reasonably well for the bladeenc and bzip2 sample

\(^3\)In the context of this thesis the term average refers to the arithmetic mean of a set of values.
programs but shows a significant increase in the misprediction rate for the gcc program, which is more complex in terms of functional logic.

<table>
<thead>
<tr>
<th>Program</th>
<th>Taken</th>
<th>Semi</th>
<th>BTFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc-opt</td>
<td>49.5%</td>
<td>12.2%</td>
<td>42.8%</td>
</tr>
<tr>
<td>gcc</td>
<td>42.3%</td>
<td>10.8%</td>
<td>47.9%</td>
</tr>
<tr>
<td>bladeenc</td>
<td>27.2%</td>
<td>7.6%</td>
<td>64.3%</td>
</tr>
<tr>
<td>bzip2</td>
<td>25.9%</td>
<td>8.2%</td>
<td>18.3%</td>
</tr>
</tbody>
</table>

Table 3.7: Static branch misprediction rate

The semi-static predictor (called Semi) uses a dedicated bit encoded in the branch instruction word to determine whether a branch is predicted taken or not-taken. This bit is set by the compiler using information from control flow analysis or execution profiles. Therefore, this type of branch predictor has to be supported by both the instruction set architecture and the compiler. For our sample programs, the branch prediction accuracy of the semi-static predictor is similar to that reported for the bimodal predictor.

The last static predictor, called BTFN (backward taken, forward not-taken), makes its decision based on the direction of the taken branch, i.e. backward if the target address is smaller than the address of the branch instruction or forward otherwise. A typical configuration, which has been proposed by Smith (1981), predicts backward branches as taken and forward branches as not-taken. As shown in the table, the BTFN predictor improves the misprediction rate of the simple static predictor only for the gcc-opt and bzip2 programs. It performs worse for gcc and bladeenc. Note that the semi-static and BTFN predictor configurations have been excluded from Table 3.1 because these predictors are not used later for the detailed evaluation of the branch predictor behaviour of our selected sample programs. This is due to the fact that these predictors are not implemented in the original SimpleScalar simulator suite.
3.4. DISTRIBUTIONS OF BRANCH MISPREDICTIONS

Table 3.8 provides a breakdown of the average dynamic branch misprediction rate for each dynamic predictor configuration. The combined predictor performs best among all other dynamic prediction schemes. In general, the results indicate a high variance of the misprediction rate among both sample programs and predictor configurations. For example, all two-level predictors show an increase of the misprediction rate of more than 30% between the \texttt{gcc} and \texttt{gcc-opt} samples just because \texttt{gcc-opt} uses additional program logic and consequently executes a higher percentage of the branch instructions in the code.

<table>
<thead>
<tr>
<th>Program</th>
<th>bimod</th>
<th>comb</th>
<th>GAg</th>
<th>GAp</th>
<th>gshare</th>
<th>PAg</th>
<th>PAp</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc-opt</td>
<td>12.2%</td>
<td>7.5%</td>
<td>13.3%</td>
<td>7.9%</td>
<td>12.4%</td>
<td>13.5%</td>
<td>12.0%</td>
</tr>
<tr>
<td>gcc</td>
<td>12.3%</td>
<td>5.7%</td>
<td>9.8%</td>
<td>5.9%</td>
<td>9.3%</td>
<td>10.3%</td>
<td>9.1%</td>
</tr>
<tr>
<td>bladeenc</td>
<td>7.1%</td>
<td>4.7%</td>
<td>6.4%</td>
<td>5.7%</td>
<td>6.1%</td>
<td>5.0%</td>
<td>4.8%</td>
</tr>
<tr>
<td>bzip2</td>
<td>6.6%</td>
<td>5.7%</td>
<td>6.3%</td>
<td>5.7%</td>
<td>6.3%</td>
<td>6.6%</td>
<td>6.4%</td>
</tr>
</tbody>
</table>

Table 3.8: Dynamic branch misprediction rate

Figure 3.2 on the next page shows for each of the four sample programs the distribution of misprediction rates over all executed branch instructions. The box-and-whisker plots (box-plots for short) used in this figure provide a graphical representation of the first quartile, the median, the third quartile, and the inter-quartile range of the misprediction rates. The twenty-fifth percentile is called the \textit{first quartile} (Q1) or the \textit{lower quartile} and the seventy-fifth percentile is the \textit{third quartile} (Q3) or \textit{upper quartile}. The vertical dimension of the central box in a plot represents the \textit{inter-quartile range}, i.e. the middle 50% of the misprediction rates. It is obtained by subtracting the first quartile from the third quartile. The \textit{median}\textsuperscript{4} of the misprediction rates is shown as a horizontal line within the box. The horizontal lines at both ends of the dashed vertical line are called the \textit{box}

\textsuperscript{4}The median is the \textit{middle value} of a list of data points. It is the smallest data point such that at least half the data points in the list are no greater than it. The median should not
Figure 3.2: Box-plots of mispredictions per branch predictor
3.4. DISTRIBUTIONS OF BRANCH MIS_PREDICTIONS

whiskers and extend to the most extreme data points which are no more than 150% of the inter-quartile range from the boundaries of the box. The values outside this range are displayed as separate small circles.

In addition to the box-plots the small diamonds show the average branch mispredictions rate, i.e. the total number of mispredictions divided by the total number of dynamic branch instructions, for a particular branch predictor configuration. The small point-down triangles show the relative number of static branches in percent (right axis) that have a misprediction rate of less than or equal to 10%. Similarly, the point-up triangles provide the ratio of static branches with a misprediction rate of less than 50%. These values are particularly interesting because they indicate that for all sample programs and dynamic branch predictor configurations more than 70% of all static branch instructions are predicted with an accuracy of at least 50%.

We can observe from the box-plots that the misprediction rates of the individual branch instructions distribute over a wide range of values. In general, the width of the distribution differs for the bimodal, combined, and two-level adaptive predictors. The bimodal and combined predictors have the least variance among all branch predictor configurations for the bladeenc and bzip2 sample programs. The branches in the gcc sample with its more complex functional logic seem to take advantage of the global branch history register that is used by the GAp predictor to index a per-address pattern history table (PHT).

In summary, the simulation results obtained from SimpleScalar running our set of sample programs suggest that the dynamic branch behaviour has to be further examined in order to explain the variance in the branch misprediction rates. The question is now whether we can distinguish between different types of branch behaviour such that we are able to analyse the predictability of a branch...
in more detail. We will answer this question in the course of the following section.

## 3.5 Branch Classification Schemes

The purpose of *branch classification* is to partition the branch instructions of a program into a number of distinct sets, called *branch classes*, such that branches with similar execution characteristics share the same branch class. The classification can be done for static as well as dynamic branch instructions. However, in the context of static analysis for branch predictors we are only interested in static classification schemes because the set of dynamic instructions cannot be determined completely at compile-time.

### 3.5.1 Static Branch Classes

Chang (1997) propose a branch classification scheme that partitions branch instructions into classes according to their dynamic behaviour. The partitioning is done statically and is based on the dynamic taken rate $pr(br)$ of the branches, as defined in Table 3.9 on the facing page. Chang refers to the SC3 and SC4 as *mixed-direction branches* and to the other branch classes as *mostly-one-direction branches*. The aim of his classification approach is to improve the accuracy of branch prediction by constructing hybrid branch predictors that select the most suitable predictor component for each branch class.

Table 3.10 on the next page lists how the static branch instructions of the four sample programs distribute over the six static branch classes. Interestingly, for all sample programs about 70% of all static branch instructions are highly biased towards either the *not-taken* (class SC1) or *taken* (class SC6) direction. In general, Figure 3.3 on page 96 shows that the average misprediction rate for all dynamic branch predictors is lower for branch instructions biased towards
3.5. BRANCH CLASSIFICATION SCHEMES

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC1</td>
<td>$0% \leq pr(br) \leq 5%$</td>
</tr>
<tr>
<td>SC2</td>
<td>$5% &lt; pr(br) \leq 10%$</td>
</tr>
<tr>
<td>SC3</td>
<td>$10% &lt; pr(br) \leq 50%$</td>
</tr>
<tr>
<td>SC4</td>
<td>$50% &lt; pr(br) \leq 90%$</td>
</tr>
<tr>
<td>SC5</td>
<td>$90% &lt; pr(br) \leq 95%$</td>
</tr>
<tr>
<td>SC6</td>
<td>$95% &lt; pr(br) \leq 100%$</td>
</tr>
</tbody>
</table>

Table 3.9: Static branch classes

either the not-taken (SC1) or taken (SC6) direction. However, the distribution of misprediction rates is significantly wider for the two-level adaptive predictors than for the bimodal or combined predictor. This is particularly true for those two-level predictors that use global history.

<table>
<thead>
<tr>
<th>Program</th>
<th>SC1</th>
<th>SC2</th>
<th>SC3</th>
<th>SC4</th>
<th>SC5</th>
<th>SC6</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc-opt</td>
<td>37.8%</td>
<td>2.1%</td>
<td>13.2%</td>
<td>11.8%</td>
<td>2.5%</td>
<td>32.5%</td>
</tr>
<tr>
<td>gcc</td>
<td>37.1%</td>
<td>1.4%</td>
<td>9.4%</td>
<td>10.4%</td>
<td>1.6%</td>
<td>40.0%</td>
</tr>
<tr>
<td>bladeenc</td>
<td>34.5%</td>
<td>2.1%</td>
<td>12.8%</td>
<td>13.9%</td>
<td>3.7%</td>
<td>33.0%</td>
</tr>
<tr>
<td>bzip2</td>
<td>37.9%</td>
<td>0.6%</td>
<td>14.6%</td>
<td>13.0%</td>
<td>1.7%</td>
<td>32.2%</td>
</tr>
</tbody>
</table>

Table 3.10: Distribution of static branch instructions over branch classes

For each branch predictor configuration defined earlier in Table 3.1 on page 84, Figures A.1, A.2, and A.3 in Appendix A show the distributions of branch mispredictions for the gcc-opt, gcc, and bzip2 sample programs, respectively. Figure 3.3 shows the distribution of misprediction rates obtained by the simulation of the bladeenc sample program. In this figure the simulation results for the PAg and static predictors are omitted. Again, the ratio in percent of static branch instructions that have a misprediction rate of less than or equal to 10% is indicated by small point-down triangles. Similarly, the point-up triangles provide the ratio
CHAPTER 3. EMPIRICAL EVALUATION OF BRANCH PREDICTORS

Figure 3.3: Box-plots of mispredictions per static class (bladeenc) of static branches with a misprediction rate of less than 50%. For the bimodal predictor 88.6% and 84.5% of all static branch instructions in the SC1 and SC6 class, respectively, have a misprediction rate of less than or equal to 10%. In contrast, only 6.4% and 8.5% of the branches in the SC3 and SC4 classes, respectively, are within the same misprediction rate range. These figures are less significant for the two-level adaptive predictors.

Furthermore, the dimensions of the individual boxes indicate that the misprediction rate for the two-level adaptive predictors are distributed over a much wider range than for the bimodal or combined predictor. We can summarise from the simulation results that branch instruction in the SC1, SC2, SC5, and SC6 class are easy to predict while most branches in the SC3 and SC4 classes are hard to predict.
A disadvantage of the static branch classification scheme proposed by Chang is that it does not address the number of times a branch actually changes its direction. For example, a branch instruction with a 50% taken rate, thus falling into the SC3 class, can either exhibit an alternating behaviour or it can first be not-taken a number of times and then be taken subsequently. While the former scenario represents worst-case behaviour the latter scenario involves only a single misprediction. Thus, branches falling into the SC3 or SC4 classes are not necessarily hard to predict. This disadvantage is addressed by the classification scheme that we will present in the next section.

### 3.5.2 Repeating Patterns

Table 3.11 defines a set of repeating patterns to capture the execution behaviour of branch instructions. These patterns are based on earlier work done by Evers et al. (1998).

<table>
<thead>
<tr>
<th>Branch Type</th>
<th>Pattern</th>
<th>Period</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taken Biased</td>
<td>$T^n$</td>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>Not-Taken Biased</td>
<td>$N^n$</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>Alternating</td>
<td>$(TN)^n \lor (NT)^n$</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>For-Type</td>
<td>$T^nN, n &gt; 1$</td>
<td>$n + 1$</td>
<td>L</td>
</tr>
<tr>
<td>While-Type</td>
<td>$N^nT, n &gt; 1$</td>
<td>$n + 1$</td>
<td>L</td>
</tr>
<tr>
<td>Simple Pattern</td>
<td>$N^mT^n, n, m &gt; 1$</td>
<td>$n + m$</td>
<td>S</td>
</tr>
<tr>
<td>Complex Pattern</td>
<td>otherwise</td>
<td>5...$\infty$</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 3.11: Branch types based on repeating branch execution patterns

The keys in this table are used in Table 3.12 on the next page, which shows the branch misprediction rate for each repeating pattern and for each sample program. The patterns represent dynamic execution behaviour of branches, thus,
it is possible that the pattern classification may change for different input data. The branch type classification is done by a Perl script that translates the branch behaviour of each static branch instruction in the trace file into a pattern string, e.g. \(\langle TNNTN \rangle\). Then, Perl's pattern matching feature determines the branch type of the pattern string.

The results of the pattern classification shown in Table 3.12 indicate that more than 50% of all static branch instructions never change their branch direction during execution and thus are not difficult to predict. Branches that alternate their direction with each execution instance represent worst-case prediction behaviour of a bimodal predictor. However, for our sample programs less than 4% of the static branch instructions exhibit this kind of behaviour. One fourth of all static branch instructions in the \texttt{gcc} sample program and about one third of the branches in the remaining three sample programs exhibit complex branch behaviour.

<table>
<thead>
<tr>
<th>Program</th>
<th>A</th>
<th>C</th>
<th>L</th>
<th>N</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc-opt</td>
<td>2.3%</td>
<td>35.4%</td>
<td>3.3%</td>
<td>32.2%</td>
<td>0.8%</td>
<td>26.0%</td>
</tr>
<tr>
<td>gcc</td>
<td>1.8%</td>
<td>25.0%</td>
<td>4.4%</td>
<td>33.3%</td>
<td>1.0%</td>
<td>34.5%</td>
</tr>
<tr>
<td>bladeenc</td>
<td>3.2%</td>
<td>33.7%</td>
<td>8.7%</td>
<td>29.0%</td>
<td>1.7%</td>
<td>23.7%</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.6%</td>
<td>32.3%</td>
<td>9.4%</td>
<td>30.0%</td>
<td>0.9%</td>
<td>25.8%</td>
</tr>
</tbody>
</table>

Table 3.12: Distribution of static branch instructions over branch types

Figure 3.4 on page 100 presents, for various branch predictor configurations, the simulation results for the \texttt{bladeenc} sample program. Each chart shows the statistical distribution of the branch misprediction rate over the six branch types defined in Table 3.11 on the preceding page. Figures B.1, B.2, and B.3 in Appendix B present similar simulation results for the \texttt{gcc-opt}, \texttt{gcc}, and \texttt{bzip2} sample programs, respectively. These figures show that for the alternating case all sample programs achieve a misprediction rate of less than 50%, although the
3.5. BRANCH CLASSIFICATION SCHEMES

A worst-case assumption would suggest a misprediction rate near 100%. This is due to the following two reasons:

- Firstly, in order to exhibit worst-case performance the two-bit counter has to toggle between its weakly taken and weakly not-taken states. The simulation results indicate that this worst-case scenario is rarely the case for the sample programs. If the counter state toggles between the strongly taken and weakly taken states the misprediction rate is reduced to 50%. Similar considerations apply for the combination of strongly not-taken and weakly not-taken states.

- Secondly, two-level adaptive predictors use patterns to record the branch history and the worst-case scenario can only occur if several branches share the same branch history pattern. This is also supported by the simulation results for the two-level and combined predictor configurations.

We can also observe from Figure 3.4 on the next page that a large number of branches classified as either N, T, or L are predicted accurately by the bimodal branch predictor. For the bimodal predictor, 60.1% of all static branches classified as L have a misprediction rate of less than or equal to 10%. In contrast, the 50% inter-quartile range for the two-level predictors is much wider.

The drawback of this classification scheme is that the classification of the branch instructions into the various branch types is not always clear-cut. Also, the scheme is based on dynamic behaviour patterns, which do not always unambiguously correspond to a specific construct in the source code, i.e. the semantic context of the branch. For example, a branch instruction that is classified as having loop-type behaviour is not necessarily associated with a loop construct in the source code because it is possible that a conditional statement also exhibits any of the loop-type branch patterns defined in Table 3.11 on page 97. On the other hand, the branch in a loop iterating only twice is classified as an alternating-type
branch. As far as the empirical analysis of branch behaviour is concerned, this drawback does not limit the validity of our simulation results.

### 3.6 Evaluation of Potential Pessimism

The previous sections in this chapter have analysed the branch predictor performance in terms of mispredictions. However, the accuracy of the branch prediction scheme has also an effect on the performance of the instruction pipeline. This section illustrates that this effect is quite significant.

Table 3.13 on the next page shows the impact of branch predictor performance on *instructions per clock cycle* (IPC). This measure can be used to directly com-
3.6. EVALUATION OF POTENTIAL PESSIONISM

<table>
<thead>
<tr>
<th>Program</th>
<th>In-order Issue</th>
<th>Out-of-order Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bimod  wc  Pessim.</td>
<td>bimod  wc  Pessim.</td>
</tr>
<tr>
<td>gcc-opt</td>
<td>0.6136  0.4469  37.3%</td>
<td>0.9569  0.5593  71.1%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.6241  0.4411  41.5%</td>
<td>0.9915  0.5462  81.5%</td>
</tr>
<tr>
<td>bladeenc</td>
<td>0.7653  0.5388  42.0%</td>
<td>1.5660  0.7714  103.0%</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.7470  0.5245  42.4%</td>
<td>1.6275  0.7471  117.8%</td>
</tr>
</tbody>
</table>

Table 3.13: Impact of branch prediction on IPC

pare the performance of an instruction pipeline for different configurations as long as the number of committed instructions remains constant. A higher IPC value means a better throughput of the instruction pipeline. The theoretical IPC, i.e. the IPC of an ideal instruction pipeline without any pipeline stalls, for a single-issue microprocessor is equal to one. The inverse of IPC is called clock cycles per instruction (CPI).

In table Table 3.13, for each sample program the simulated IPC is shown for a bimodal predictor and for a worst-case predictor, i.e. a branch misprediction penalty is associated with each branch instruction. We assume a branch misprediction penalty of three clock cycles, which represents a reasonable value for a five-stages instruction pipeline as used by SimpleScalar. The results are given for an in-order and out-of-order instruction issue pipeline, respectively, and were obtained from the cycle-level simulator sim-outorder of the SimpleScalar tool set. The simulated pipeline can issue up to four instructions to the instruction fetch queue with each clock cycle, i.e. the theoretical IPC is equal to four. As far as pipeline utilisation is concerned, in-order instruction issue is in general less efficient than out-of-order issue because pipeline stalls due to pipeline hazards cannot be reduced by rearranging the sequence of instructions dynamically. This is shown in the simulation results by the lower IPC of the in-order pipeline for all sample programs.

101
The third column (Pessim.) for each of the two pipeline configurations provides the increase in percent of the number of clock cycles required for executing the sample programs relative to the simulation result for a bimodal predictor configuration. This value represents the amount of pessimism introduced by the assumption that each branch instruction results in a misprediction and it is given by:

\[
pessim = \frac{CPI_{wc} - CPI_{bimod}}{CPI_{bimod}} \cdot 100\% = \left( \frac{IPC_{bimod}}{IPC_{wc}} - 1 \right) \cdot 100\% \quad (3.1)
\]

The simulation results shown in Table 3.13 on the previous page illustrate the significant level of pessimism associated with this assumption. We can observe that the most significant difference in performance occurs for the bzip2 and bladeenc sample programs using an out-of-order issue pipeline. For these two sample programs the amount of pessimism is more than 100% compared to that of a simple bimodal predictor because of its high prediction accuracy.

Also, the number of branch instructions that have a misprediction rate of 100% is very small. For the bladeenc sample program using the bimodal predictor only about 2% of the static branch instruction exhibit this kind of worst-case branch predictor performance. These branch instructions account for 50,946 dynamic branch instances of a total of 75,280,109 branch instances. The average number of instances for these branches is about 435. However, the majority of these mispredictions is caused by only two branch instructions in the bladeenc sample program, each being executed 23,424 times. The case study in Section 7.3 on page 221 will address how the source code of bladeenc can be modified in order to eliminate these two alternating branch instructions.

3.7 Summary

This chapter has evaluated the branch prediction behaviour of various sample programs by using the SimpleScalar microprocessor simulator tool set. The sam-
ple programs were simulated with seven dynamic and one static branch predictor. The simulation results have shown that there is a significant amount of variance in the misprediction rate among the individual branch instructions of a program. Also, the prediction accuracy varies both between the individual branch predictors and the sample programs.

In order to provide for a finer granularity of branch predictor analysis we have presented two static branch classification schemes, which classify branch instructions statically according to their dynamic execution behaviour. The first classification scheme uses the dynamic taken rate of a branch instruction to categorise it into one of six static branch classes while the second scheme is based on a set of six dynamic branch behaviour patterns. We have used both classification schemes to evaluate the branch prediction behaviour of the sample programs.

The simulation results presented in this chapter have indicated that only a very small number of branch instructions have a misprediction of 100%. Therefore, simply assuming a branch misprediction for each branch results in a significant level of pessimism. The amount of pessimism caused by this assumption varies between 71.1% and 117.8% for an out-of-order instruction pipeline. We can conclude from these results that accurate static timing analysis of dynamic branch predictors is paramount to calculating tight WCET estimates.
Chapter 4

Static Analysis of Bimodal Branch Predictors

The purpose of this chapter is to develop a method for static analysis of dynamic branch predictors that use a branch history table (BHT). This method aims at reducing the overestimation associated with WCET analysis for microprocessors using such branch prediction schemes. Many existing WCET analysis methods do not provide a means for analysing branch prediction and just make the conservative assumption that all branches are predicted incorrectly. The experimental results presented in the previous chapter have shown that this assumptions leads to a gross overestimation of the WCET for most branch instructions. Consequently, a large amount of the available processing resources remains unused because the estimated microprocessor utilisation is too high.

The underlying idea of the method presented here is to transform the basic approach of the data cache analysis method proposed by Lundqvist and Stenström (1999a) to the area of static WCET analysis for dynamic branch predictors. Lundqvist and Stenström classify memory accesses as either predictable or unpredictable based on whether the associated memory address is known statically or not. Similarly, we classify branch instructions as either easy-to-predict
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

or hard-to-predict, based on their classification criterion as to whether or not the behaviour of a branch instruction can be predicted using information available at compile-time.

In the context of this chapter we will concentrate on a bimodal branch predictor using a two-bit saturating up-down counter, as described earlier in Section 2.2 on page 46.

In this chapter, Section 4.1 introduces the basic terminology regarding basic blocks and control flow graphs that will be used throughout this chapter. In addition, this section defines the notion of WCET for the context of basic blocks and paths using a simplified but yet sufficiently accurate model describing the execution time behaviour of an instruction pipeline. A more detailed instruction model will be presented in Chapter 5, Integration with Pipeline Analysis. Section 4.2 defines a theoretical model of the behaviour of the bimodal branch predictor that is used as reference throughout this chapter. Section 4.3 provides the foundation of the classification approach for branch instructions presented in this chapter. Section 4.4 establishes the correspondence between the branch classification model and basic control flow statements of the C programming language such as loops and conditional construct. For these basic control flow statements, analysis models for estimating the worst-case number of branch mispredictions are defined. The static analysis approach itself is independent from the programming language as it is performed on the control flow graph and basic blocks derived from the compiled code. Section 4.5 illustrates the use of the analysis models through a case study evaluating the behaviour of a code example with three different execution scenarios. Section 4.6 discusses the effect of branch interference, i.e. several branch instructions competing for the branch predictor entry, on the number of mispredictions. Simulation results based on the evaluation environment presented earlier in Chapter 3 show what impact the size of a branch history table can have on the effect of branch interference. As far as the
4.1 Basic Terminology

This section introduces the basic terminology regarding basic blocks and control flow graphs that will be used throughout this chapter. In addition, this section defines the notion of WCET for the context of basic blocks and paths using a simplified but yet sufficiently accurate model describing the execution time behaviour of an instruction pipeline. The level of detail of this model is sufficient to determine upper bounds on the number of branch mispredictions. In Chapter 5 the model is extended such that it accounts for instructions pipelines and the impact of branch mispredictions on the behaviour of the instruction pipeline.

Figure 4.1 depicts the two basic alternatives for determining the number of mispredictions that can be observed for a given branch predictor configuration. The bottom half of this figure shows the work-flow of the execution driven approach for obtaining the number of mispredictions. The approach taken in this thesis for the static analysis of branch predictors is shown in the upper half of Figure 4.1.

![Figure 4.1: Determining the performance of branch predictors](image)

Basically, the following three criteria can be used for the classification of a branch
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

1. Code Structure. This criterion captures the semantic context of a branch within a program, e.g. whether the branch belongs to an if-then-else or a loop construct.

2. Dynamic Branch Behaviour. The dynamic execution pattern of a branch is taken into account by this criterion, e.g. whether a branch is biased toward taken or not-taken, or exhibits any particular pattern.

3. Branch Predictor Configuration. This criterion identifies whether the branch predictor uses local or global history to make its predictions. With a global history predictor the behaviour of other branches may influence the prediction of the branch outcome (branch correlation) and, therefore, the scope of analysis is much wider than for a local history predictor.

The first two criteria are defined by the branch instruction itself while the third criterion is a property of the branch prediction scheme. With a given prediction scope, the dynamic branch behaviour directly influences the prediction taking into account a given branch predictor configuration. However, it is not always possible to determine the behaviour of a branch statically at compile-time. Therefore, the semantic context of a branch, which is a static property known at compile-time, is used instead to make assumptions about the dynamic branch behaviour. For example, the branch instruction that evaluates the condition in a for-loop construct with n iterations is taken n – 1 times and not-taken once.

The static analysis method presented in the course of this chapter uses the semantic context of a branch as a criterion for the classification of branch instructions. This requires that we first introduce a definition of the term semantic context:
4.1. BASIC TERMINOLOGY

Definition 4.1 (Semantic Context)

*The semantic context of a branch instruction is defined by the structure of the control flow graph into which the branch is embedded.*

It is important to note that the control flow graph has to be analysed on assembler instruction level rather than on high-order language (HOL) level, because the structure of the generated control flow graph not only depends on the source code itself but also depends on the compiler used and other factors, such as data types and compiler optimisation settings.

4.1.1 Basic Block and Control Flow Graph

Definition 4.2 (Basic Block)

*A basic block* $b_i$ *of a program is the longest continuous sequence of low-level instructions in memory such that the basic block is always entered at its first instruction and always exited at its last instruction.*

The following properties of basic blocks can be observed from this definition:

- Each basic block contains at most one branch instruction, which is then the last instruction of the basic block.

- If a basic block does not contain a branch instruction then the first instruction of the subsequent block is a branch target.

- If any instruction in the basic block is executed then this implies that all instructions are executed.

Based on the notion of basic blocks we can define the *control flow graph* of a program as follows.
Definition 4.3 (Control Flow Graph)

The control flow graph of a program is a directed graph \( G(V, E) \), where \( V \) is the finite non-empty set of vertices (basic blocks) and \( E \subseteq (V \times V) \) is the set of directed edges, i.e. the possible control flow transitions between the basic blocks.

The ordered pair \((b_i, b_j) \in E\), which may also be denoted as \(b_i \rightarrow b_j\), represents the transition between two basic blocks \(b_i, b_j \in V\). A branch block is a basic block that contains a conditional branch instruction. Only branch blocks can change the control flow of a program depending on the outcome of a condition. The set of all branch blocks is called the subset \(V_c \subseteq V\). Each branch block has two outgoing transitions while all other basic blocks have a single outgoing transition. We distinguish between the two transitions of a branch block by saying that the transition \((b_i \overset{T}{\rightarrow} b_j) \in E\) represents the taken path and the transition \((b_i \overset{N}{\rightarrow} b_k) \in E\) the not-taken path of the branch instruction in \(b_i \in V_c\). For the remainder of this thesis we will always use \(N\) and \(T\) to denote a not-taken or taken branch, respectively. Furthermore, we use \(b_i \in V_c\) to denote the branch instruction contained in this branch block.

A path \(p\) of \(G\) is a finite sequence \(\langle b_1, \cdots, b_k \rangle\) of basic blocks \(b_i \in V\) such that \((b_i \rightarrow b_{i+1}) \in E\). A path is said to form a circle if the first and the last basic block of the path are the same. In the context of programming languages, a circle represents a loop. We explicitly allow a circle that contains only a single basic block, which in this case is a branch block with the transitions \((b_i \overset{T}{\rightarrow} b_i) \in E\) and \((b_i \overset{N}{\rightarrow} b_{i+1}) \in E\). The path obtained by the concatenation of a cycle \(c\) with itself is denoted \(c^n\), where \(n\) is the number of times \(c\) is repeated. This concatenated path \(c^n\) represents the execution of the cycle \(c\) over \(n\) iterations, and each basic block in \(c\) is executed for \(n\) instances.

Figure 4.2 shows an example of a control flow graph and its corresponding dominator tree for a loop construct with an embedded break statement. In this example, basic blocks \(b_1\) and \(b_5\) represent the start and end node of the CFG,
4.1. BASIC TERMINOLOGY

(a) Control Flow Graph  (b) Dominator Tree

Figure 4.2: Example of a loop construct with break statement
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

respectively. The set of all branch blocks is $V_c = \{b_1, b_3, b_4\}$. All edges $b_i \rightarrow b_j$, with $b_i \in V_c$, are marked with their branch direction, which is either taken (T) or not-taken (N). All other edges do not leave from branch blocks and are therefore just marked with N. The path $\langle b_3, b_4 \rangle$ forms a circle following the transitions $(b_3 \stackrel{N}{\rightarrow} b_4), (b_4 \stackrel{T}{\rightarrow} b_3)$.

4.1.2 Static and Dynamic Instructions

In the context of this thesis we distinguish between static instructions and dynamic instructions. These terms are defined by the following two definitions.

**Definition 4.4 (Static Instruction)**

Static instructions encompass all assembler instructions defined in the object code of a program. All static instructions can be uniquely identified by their instruction address.

The sequential execution of a program creates an execution trace of the instructions. This leads to the following definition:

**Definition 4.5 (Dynamic Instruction)**

Dynamic instructions are instances of static instructions created during the execution of a program.

In particular, some static instructions may be executed several times or not even at all. For the context of our theoretical model we restrict execution traces to branch instructions only as these are the only instructions affected by and affecting the behaviour of a branch predictor.

Let $B_1, B_2, \ldots, B_p$ be the sequence of dynamic branch instructions, i.e. execution trace, of a program $P$ executed in sequential order. This execution trace comprises instances of all static branch instructions being executed during a particular program run. Each dynamic branch instruction $B_i$ in the execution trace
4.1. BASIC TERMINOLOGY

has an outcome \(d_i\), which is either \textit{taken} or \textit{not-taken}. For every dynamic branch instruction \(B_i\) in the execution trace, we write \(addr(i)\) to denote the address of the static branch instruction in \(P\) associated with \(B_i\). Each static branch instruction is uniquely identified by the basic block it is located in because basic blocks may contain at most one branch instruction. We identify a static branch instruction by the basic block \(b_i \in V_e\) in which it is located.

We transfer the behaviour of branches and branch predictors to the theory of formal languages and automata. In this context, the concatenation of the individual branch outcomes \(d_i\) can be interpreted as a string \(w = \langle d_1d_2\ldots d_p \rangle\) over the alphabet \(\Sigma = \{N, T\}\). This string represents the global branch history of a single execution scenario of the program. Strings with an arbitrary but fixed length are also called \textit{patterns}. In this chapter, however, we will only model the behaviour of bimodal branch predictors, which predict the outcome of a branch instruction using a saturating two-bit counter indexed by the branch address. Thus, the single string \(w\) representing the outcomes of all branches in the global execution trace has to be broken down into separate strings \(w_i\) for each static branch instruction \(b_i\) being executed.

A number of different execution traces may exist for a program and the branch behaviour of each execution trace is described by a separate string. If we consider now all possible execution traces, the behaviour of a static branch instruction is completely described by the collection of strings for this particular branch. This collection can be expressed by a language \(L(b_i)\), which is given by:

\[
L(b_i) = \bigcup_{j \in P} \{w_i^j\},
\]

where \(P\) is the set of all possible execution scenarios of the program \(P\).

It should be noted that although the collection of all languages over \(\Sigma\) is uncountable in general, the language \(L(b_i)\) has a finite number of strings if the program
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

terminates for all possible execution scenarios – which we assume throughout this thesis. The fact that a program eventually terminates in combination with its functional correctness establishes the total correctness of a program. However, we will not further consider the proof of total correctness of programs in this context. This constraint is also required in order to obtain a finite WCET figure.

An overview of branch prediction techniques and in particular the characteristics of bimodal branch predictors will be presented in the following two sections.

4.1.3 Worst-Case Execution Time for Basic Blocks

This section defines the notion of WCET for the context of basic blocks using a simplified instruction pipeline model. The timing schema originally proposed by Shaw (1989) is used for low-level timing analysis of basic blocks and is modified to include the timing effects caused by branch mispredictions. Without loss of generality, we measure the WCET of a basic block in clock cycles rather than in a continuous time measure.

We denote the WCET in clock cycles of a basic block $b_i$ by $T(b_i)$. If $cycle_{start}$ and $cycle_{end}$ are the clock cycles when the first instruction of basic block $b_i$ has been fetched and the last instruction has been retired, respectively, then $T(b_i)$ can be calculated as follows:

$$T(b_i) = \max_{j \in P} (cycle_{j\ start} - cycle_{j\ end} + 1), \quad (4.2)$$

where $P$ is the set of all possible instruction pipelining scenarios, including effects such as pipeline hazards and instruction/data cache accesses but excluding branch mispredictions.

Excluding the effects of branch misprediction from Equation (4.2) is because branch mispredictions interrupt the instruction fetch mechanism of the microprocessor resulting in a gap of several clock cycles between the basic block with
4.1. BASIC TERMINOLOGY

the mispredicted branch instruction and the branch target block. Therefore, we associate the timing effects of branch mispredictions with the WCET of a path \( p = \{b_1, b_2\} \), with \( b_1 \in V_c, b_2 \in V \), and \( (b_1 \rightarrow b_2) \in E \), by defining the following equation:

\[
\tilde{T}(p) = T(b_1) + T(b_2) + \delta,
\]

where \( \delta \geq 0 \) is the branch misprediction penalty, i.e. the number of clock cycles required by the instruction pipeline to recover from a branch misprediction. Note that throughout this thesis, we will write \( \tilde{T}(p) \) to denote the WCET of a path \( p \) including branch misprediction penalties.

It should be noted that Equation (4.3) combines the execution time of two adjacent basic blocks without taking into account the potential overlap among these blocks in the instruction pipeline. Although this simplistic instruction pipeline model imposes significant pessimism on the final WCET estimate it is sufficient to illustrate our static analysis approach for dynamic branch predictors in this chapter. This is because our approach provides bounds on the maximum number of branch mispredictions and is therefore independent from the actual method used for obtaining execution time estimates for basic blocks.

We defer the detailed modelling of the execution time behaviour of an instruction pipeline for the purpose of WCET analysis to Chapter 5. There, the overlap provided by instruction pipelines will be included to reduce the pessimism of WCET estimate and we will describe the integration of an extended instruction pipeline analysis model with our static analysis approach for dynamic branch predictors presented in the following sections.
4.1.4 Basic Assumptions

With respect to other areas of WCET analysis we make the following assumptions:

1. **Program Flow Analysis**: A list of possible program execution paths and upper bounds on the number of loop iterations (also called loop bounds) are available from program flow analysis. Such information is either provided by the programmer as manual program annotations (Chapman et al., 1994) or is determined automatically (Healy et al., 1998).

2. **Instruction Pipeline Analysis**: Information about the execution time behaviour of basic blocks is available from instruction pipeline analysis. For the purpose of the analysis approach presented later in this chapter, the level of granularity of providing execution time figures for basic blocks rather than individual instructions will be sufficient. However, instruction pipeline analysis has to be able to distinguish between execution times for the mispredicted and correctly predicted case of a branch instruction inside a basic block.

3. **Branch Interference**: Different branch instructions do not interfere with each other, for example, by relocating branch instructions in the object code such that no two branches share the same predictor state entry in the BTB. The effect of branch interference in general and the rationale for the assumption we make here is discussed in more detail in Section 4.6 on page 158.

4.2 Bimodal Branch Predictors

Based on our interpretation of branch execution traces as strings over the alphabet $\Sigma = \{N,T\}$ we represent the two-bit saturating counter of a bimodal
branch predictor by a deterministic finite automaton (DFA), whose states and transitions are depicted in Figure 4.3. This saturating up-down counter is similar to the one originally used in the branch target buffer proposed by Smith (1981). The counter records the branch history of recent instances of a certain branch instruction. The records are then used to predict the outcome of the next instance of the branch. If the branch is taken the counter is incremented and the counter is decremented when the branch is not-taken. The next instance of the branch is predicted according to the most-significant bit of the counter, i.e. the branch will be predicted taken when the bit is set and predicted not-taken otherwise, as shown in Figure 4.3(b).

![State Transitions and Predictor States](image)

**Figure 4.3**: States and transitions for a two-bit saturating counter

For each static branch instruction $b_i$, we write $\text{lookup}(i)$ to denote the two-bit saturating counter associated with that branch. An ideal predictor maps a unique predictor state to each branch instruction, i.e. the mapping between branch instructions and predictor states is injective. Due to hardware space limitations within a microprocessor an injective mapping is not feasible for real branch predictor implementations as it would require a branch predictor state entry for each single address in the microprocessor’s address space. The implications of this practical limitation will be discussed in more detail in Section 4.6.

Furthermore, we denote the predictor state prior to resolving the $j$th instance of a conditional branch instruction as $q_j$. The predicted direction of the branch is
determined by the state $q_j$ and denoted by the mapping $\text{dir} : Q \mapsto \Sigma$. After the branch is resolved, its outcome $d_j \in \Sigma$ is used to update the state of the branch predictor. A branch misprediction occurs if the predicted and resolved outcome of the branch do not match, i.e. $\text{dir}(q_j) \neq d_j$, otherwise the branch has been predicted correctly. The new predictor state $q_{j+1}$, which will be used to predict the outcome of the $(j+1)$th instance of the branch, is determined by the state transition function $\delta : Q \times \Sigma \mapsto Q$

$$q_{j+1} = \delta(q_j, d_j)$$ (4.4)

### 4.2.1 Initial Branch Predictor State

The initial state $q_0$ of the counter is typically either the weakly taken or the weakly not-taken state. The weakly not-taken state may give a little advantage to branches that follow the backward taken, forward not-taken (BTFN) prediction pattern, as discussed earlier in Chapter 3. Forward branches usually represent logical conditions in a program. In general, however, the initialisation policy may vary from one microprocessor to another. The SimpleScalar microprocessor simulator, for example, initialises each entry in the branch history table by alternating between the weakly taken and weakly not-taken states.

In practice, however, we do not know whether or not a particular predictor state has already been altered by another branch instruction. We therefore interpret the initial state of a branch predictor in a more general context and define it as the state prior to applying a string to the predictor state. Thus, through the state transition function $\delta$ the string $w$ translates the initial state $q_0$ into the final state $q_n$, formally:

$$q_n = \delta(q_0, w),$$ (4.5)

with $w \in \Sigma^*$, $n = |w|$, and $q_0 \in Q$. 

118
For short, we may also write \( q_0 \xrightarrow{w} q_n \) for a given predictor.

Without further knowledge about the actual initial state of the predictor we assume a state such that the number of mispredictions is guaranteed to be maximal. In other words we are seeking for a worst-case initial branch predictor state. This is formally introduced in the following section.

### 4.2.2 Number of Branch Mispredictions

The exact number of branch mispredictions is not only determined by the string \( w \in \Sigma^* \) representing the execution trace but also by the initial state of the branch predictor. We define the mapping \( mp : \Sigma^* \times Q \rightarrow \mathbb{N} \) in order to associate the number of branch mispredictions with a string and an initial predictor state. In the worst-case, the number of branch mispredictions is equal to the length of a string, i.e. all branch instructions are being mispredicted. Thus we can state the following trivial upper bound on the number of branch mispredictions:

\[
\forall w \in \Sigma^*: \quad mp(w, q_0) \leq |w|, \quad q_0 \in Q
\]

(4.6)

The initial state \( q_0 \) of the branch predictor may not be known by means of static program analysis and therefore an upper bound on the number of branch mispredictions needs to be defined independently from the actual initial predictor state. This upper bound is defined by the mapping \( \hat{mp} : \Sigma^* \rightarrow \mathbb{N} \), which is given by:

\[
\hat{mp}(w) = \max_{q \in Q} (mp(w, q)), \quad w \in \Sigma^*
\]

(4.7)

As far as WCET analysis for dynamic branch predictors is concerned, we are interested in the worst-case initial state of the predictor for a given string such that the maximum number of branch mispredictions occurs. The existence of such an initial state is defined as follows:

\[
\exists q_{wc} \in Q: \quad mp(w, q_{wc}) = \hat{mp}(w), \quad w \in \Sigma^*
\]

(4.8)
4.2.3 Basic Properties of a Bimodal Branch Predictor

As far as our WCET analysis approach presented in the following is concerned, an essential property of a bimodal branch predictor is that it does not change its state any more if more than three consecutive branches have the same outcome:

\[ w \in \{ a^i \mid i \geq 3 \} \Rightarrow \delta(q, w) = \delta(q, a^3), \]  \hspace{1cm} (4.9)

with \( a \in \Sigma \) and \( q \in Q \).

Proof: Equation (4.9) is trivially true for \( i = 3 \). Let us now assume that \( i > 3 \) and \( a = N \). We can rewrite the string \( w = \langle a^i \rangle \) as \( w = \langle uv \rangle \). The longest path without any circles through the predictor states is from the strongly taken \( ST \) to the strongly not-taken \( SN \) state by using three consecutive not-taken branches, i.e. at most three not-taken branches are needed to reach the strongly not-taken state from any state of the predictor. After reaching the strongly not-taken predictor state, any subsequent not-taken branch will no longer result in a change of the predictor state since \( \delta(SN, N) = SN \). Similar considerations apply to the case where \( a = T \). \( \Box \)

4.3 Branch Classification

The branch classification method presented in this chapter transforms an approach proposed previously by Lundqvist and Stenström (1999a) for data cache analysis to the area of WCET for dynamic branch predictors. There are a number of similarities between caches and dynamic branch predictions techniques. For example, the main problem of data cache analysis is to make data references predictable such that it can be decided whether a particular reference is present in the cache or not. While this is not possible for all data references, a relatively
large number of references turn out to be predictable. Lundqvist and Stenström show that more than 84% of the data accesses are in fact predictable based on sample programs from the SPEC95 benchmark suite. In their approach, analysis is performed for predictable data references only and unpredictable references are excluded from the cache so they do not make the contents of the cache unpredictable. Effects of being unpredictable are not only the accesses have to be assumed to be not in the cache but also that potentially valuable data is displaced. By taking Lundqvist’s approach, only the penalty for missing the cache is suffered.

Similarly to data caches, the outcome of conditional branches typically depends on input data that is not always known at compile-time. However, not only does the data reference need to be predictable in order to be able to determine the outcome of a conditional branch, but also knowledge of the data itself is necessary.

In order to gain independence from the input data determining the outcome of a branch instruction, our analysis method uses its semantic context as a criterion for the classification of branch instructions. Based on this classification, branches are either easy-to-predict or hard-to-predict in terms of static timing analysis.

The main problem associated with static analysis for dynamic branch predictors is to determine the behaviour of branches without actually executing the program. In principle, we can determine the branch behaviour statically by

- analysing the semantic context into which the branch instruction is embedded;
- applying data-flow analysis techniques to determine the possible outcome of a branch;
- using manual annotations provided by the programmer in order to define branch behaviour patterns.
In the context of this work we do not further consider data-flow analysis techniques but assume that such techniques are already available as part of the overall WCET analysis framework, and used to generate program annotations.

4.3.1 Definitions

A branch instruction is classified as being *easy-to-predict* if there exists a static branch execution pattern and this pattern can be determined from the semantic context of the branch by means of static code analysis. Otherwise, the branch instruction is classified as being *hard-to-predict*. This course-grained classification of branch instructions is summarised by the following two definitions:

**Definition 4.6 (Easy-to-predict Branch)**

An easy-to-predict branch instruction is a branch instruction whose execution behaviour can be accurately determined from its semantic context at compile-time.

The execution behaviour is described by a static branch execution pattern, which we will later use to further refine the classification scheme for easy-to-predict branches.

**Definition 4.7 (Hard-to-predict Branch)**

A hard-to-predict branch instruction is a branch instruction whose execution behaviour is dominated by the characteristics of the input data defining its branch condition and this input data can only be determined during execution.

4.3.2 Classification Based on Branch Pattern Types

Table 4.1 on the next page defines the classification scheme for *easy-to-predict* branches and establishes a set of branch pattern types for describing the execution behaviour of branch instructions. The patterns in the second column of the table
are expressed by using regular expressions in the form $w^m$, where $w \in \Sigma^*$ is said to be a basic pattern. The basic pattern represents the behaviour of the construct we want to analyse in order to derive its worst-case execution scenario. If not stated otherwise, we will use $m$ to denote the number of times the construct is repeated. For the analysis of the behaviour of loop constructs we use $n$ to define the number of loop iteration. In this case, $m$ is number of times the loop construct itself is being repeated.

The alternating (Ax) and biased (Bx) patterns represent a special case of the loop-type (Lx) patterns for $n = 2$ and $n = 1$ loop iterations, respectively. Note that we will use the terms pattern and regular expression interchangeably throughout this text. The last column of the table provides the period of each branch pattern, which is basically the length of the basic pattern. The length of a pattern $w^m$ is then equal to $m \cdot |w|$.

<table>
<thead>
<tr>
<th>Key</th>
<th>Evers</th>
<th>Branch Type</th>
<th>Pattern</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>T</td>
<td>Taken Biased</td>
<td>$T^m$</td>
<td>1</td>
</tr>
<tr>
<td>BN</td>
<td>N</td>
<td>Not-Taken Biased</td>
<td>$N^m$</td>
<td>1</td>
</tr>
<tr>
<td>AT</td>
<td>A</td>
<td>Alternating (first taken)</td>
<td>$(TN)^m$</td>
<td>2</td>
</tr>
<tr>
<td>AN</td>
<td>A</td>
<td>Alternating (first not-taken)</td>
<td>$(NT)^m$</td>
<td>2</td>
</tr>
<tr>
<td>ATT</td>
<td>-</td>
<td>Alternating (first two taken)</td>
<td>$T(TN)^m$</td>
<td>2</td>
</tr>
<tr>
<td>ANN</td>
<td>-</td>
<td>Alternating (first two not-taken)</td>
<td>$N(NT)^m$</td>
<td>2</td>
</tr>
<tr>
<td>LT</td>
<td>L</td>
<td>Loop-Type (taken biased)</td>
<td>$(T^{n-1}N)^m$, $n \geq 3$</td>
<td>$n$</td>
</tr>
<tr>
<td>LN</td>
<td>L</td>
<td>Loop-Type (not-taken biased)</td>
<td>$(N^{n-1}T)^m$, $n \geq 3$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

Table 4.1: Branch pattern types classified as easy-to-predict

The selection of these branch patterns represents the most interesting cases in terms of bimodal branch prediction and is based on earlier work done by Evers et al. (1998). The second column of the table provides the key used originally by Evers et al. and earlier in Table 3.11 on page 97. The branch types Simple
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

and Complex used by Evers et al. represent execution patterns that cannot be determined from the semantic context of the branch but only from the data used as input for the branch condition. Therefore, these two branch types are classified as being hard-to-predict and therefore not included in Table 4.1 on the previous page. A branch instruction classified as hard-to-predict is excluded from static WCET analysis and it is assumed that all instances of this branch result in a branch misprediction.

We will show in the course of this chapter that the branch execution patterns defined in Table 4.1 on the preceding page can be associated with basic control flow structures of object code generated from source code written in a high-level programming language, such as C.

4.3.3 Alternating Branch Patterns

Figure 4.4 shows the state transition diagram of a DFA with a set of states $Q = \{ST, WT, WN, SN, X\}$ that accepts strings causing a worst-case misprediction rate of the bimodal branch predictor. The automaton is designed such that its set of accept states is $F = \{WT, WN\}$ can only be reached from other states by transitions resulting in a branch misprediction. In case of a correct prediction, the state $X$ is reached from any other state and the pattern can no longer be accepted by the automaton.

Figure 4.4: DFA accepting strings with worst-case predictor behaviour

Note that the start state is not marked in the state transition diagram because we
assume that the start state $q_0$ of the DFA can be any of the states $ST$, $WT$, $WN$, and $SN$. When we apply the alternating patterns from Table 4.1 to the automaton we can see that the pattern $\langle (TN)^m \rangle$ is accepted if the start state is $WN$. Likewise, the start state $WT$ corresponds to the pattern $\langle (NT)^m \rangle$. In addition to these two alternating patterns, the patterns $\langle T(TN)^m \rangle$ and $\langle N(NT)^m \rangle$ also cause a worst-case misprediction rate if the start state is $SN$ and $ST$, respectively.

4.3.4 Loop-Type Branch Patterns

A branch instruction associated with the condition of a loop statement is called loop control branch and defined as follows:

**Definition 4.8 (Loop Control Branch)**

Let $G(V, E)$ be the control flow graph of a program and $c$ be a cycle in $G$. The branch instruction in a basic block $b_i \in (c \cap V_c)$ is called loop control branch if there exist two transitions $(b_i \rightarrow b_j) \in E$ and $(b_i \rightarrow b_k) \in E$ such that $b_j \in c$ and $b_k \notin c$. The transition $b_i \rightarrow b_k$ is an exit transition of the loop represented by $c$.

For example, the behaviour of a loop control branch with no other branches involved in the execution is defined by the language $L(B_i) \subseteq \Sigma^*$, assuming that the loop iterates five times:

$$L(B_i) = \{TTTTN\} \quad (4.10)$$

If not otherwise noted we assume that the strings in $L(B_i)$ are ordered according to their appearance in the execution trace. For an arbitrary number of loop iterations $n > 0$ the language $L(B_i)$ is given by:

$$L(B_i) = \{T^{n-1}N\} \quad (4.11)$$

Consider a loop-type branch instruction whose execution behaviour is given by the string $w = \langle TTN \rangle$, i.e. the loop iterates three times. For the worst-case
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

initial state SN, all three instances of the branch are mispredicted, therefore, $\hat{mp}(w) = 3$. If we assume $m$ repetitions of the loop construct itself the maximum number of branch mispredictions for the string $w^m$ is given by:

$$\hat{mp}(w^m) = m \cdot \hat{mp}(w) \quad (4.12)$$

However, this figure is overly pessimistic since we simply assume the worst-case initial state for each individual repetition of the loop construct. The bound on the number of mispredictions can be stated more accurately if we take into account the final state of the predictor each time the string $w$ has been executed. For our example stated above, the state transitions for the first three loop repetitions are:

$$SN \xrightarrow{w} WN \xrightarrow{w} WT \xrightarrow{w} WT$$

Based on this observation, we can now restate the upper bound on the number of branch mispredictions for the repeated execution of a loop construct assuming $m$ repetitions:

$$mp(w^m, q_{wc}) = \sum_{i=0}^{m-1} mp(w, q_i) \quad (4.13)$$

with $q_0 = q_{wc}$ and $q_i = \delta(q_{i-1}, w)$.

4.3.5 Mispredictions per Branch Pattern Type

Table 4.2 on the next page summarises the number of branch mispredictions that can be expected for each basic pattern depending on the initial state of the bimodal branch predictor. The last column in this table provides the total number of branch instances, $|w|$, in the branch pattern in order to make comparison with the number of branch mispredictions easier.
### 4.4 WCET Analysis of Basic Control Statements

In this section, we present and discuss our static analysis approach for deriving the worst-case behaviour of bimodal branch predictors. The approach is developed for different control flow constructs on MIPS PISA object code generated by the GCC compiler (Gough and Stallman, 2004) from basic control statements of the C programming language, as an example.

We perform static analysis on object code level rather than high-level source code because modern compilers, including GCC, apply various optimisations techniques during code generation with the side effect that the control flow graph on object code level may be different from the original source code.

In particular, we assume in the following discussion that a loop optimisation technique called *loop inversion* (Muchnick, 1997) has been applied by the compiler. The principle of this code transformation has been illustrated earlier in Figure 2.1 on page 42.

| Key | Pattern | SN | WN | WT | ST | $|w|$ |
|-----|---------|----|----|----|----|----|
| BT  | $T^m$   | 2  | 1  | 0  | 0  | $m$ |
| BN  | $N^m$   | 0  | 0  | 1  | 2  | $m$ |
| AT  | $(TN)^m$| $m$| $2m$| $m$| $m$| $2m$|
| AN  | $(NT)^m$| $m$| $m$| $2m$| $m$| $2m$|
| ATT | $T(TN)^m$| $2m + 1$| $m + 1$| $m$| $m$| $2m + 1$|
| ANN | $N(NT)^m$| $m$| $m$| $m + 1$| $2m + 1$| $2m + 1$|
| LT  | $(T^{m-1}N)^m$, $n = 3$| $3 + m$| $1 + m$| $m$| $m$| $m \cdot n$|
| LT  | $(T^{m-1}N)^m$, $n > 3$| $2 + m$| $1 + m$| $m$| $m$| $m \cdot n$|
| LN  | $(N^{n-1}T)^m$, $n = 3$| $m$| $m$| $1 + m$| $3 + m$| $m \cdot n$|
| LN  | $(N^{n-1}T)^m$, $n > 3$| $m$| $m$| $1 + m$| $2 + m$| $m \cdot n$|

Table 4.2: Maximum number of mispredictions for basic patterns
Listing 4.1: C source code for for-loop construct

```c
int_t for_loop(int_t *data, int_t n)
{
    int_t i;
    int_t result = 0;

    for (i = 0; i < n; i++) {
        result += data[i];
    } /* end for */

    return result;
} /* end of for_loop */
```

4.4.1 Analysis of Simple Loop Constructs

In our first example we analyse the branch predictor behaviour of a simple loop construct that executes for \( n \) iterations. Listing 4.1 shows the C source code of a function that contains a simple for-loop construct. The corresponding control flow graph is depicted in Figure 4.5.

![Control flow graph for a simple loop construct](image)

The function starts with basic block \( b_1 \) that initialises the function variables \( i \) and \( result \), and ensures that the loop bound \( n \) is greater than zero prior to entering the loop body for the first time. This initial check of the loop condition may be omitted if the compiler is able to determine from the range of the loop bound variable \( n \) that the condition is at least true once for the first loop iteration – which is not the case in this example since \( n \) can also be zero or even negative. Basic block \( b_2 \) represents the body of the loop construct and also contains the
4.4. WCET ANALYSIS OF BASIC CONTROL STATEMENTS

branch instruction that controls the iteration of the loop. We refer to this branch instruction as *loop control branch*. The static branch execution pattern for the loop control branch in basic block \( b_2 \) being executed \( n \)-times is straightforward and given by:

\[
\langle \overbrace{TT \ldots T}^{(n-1)\text{-times}} N \rangle = T^{n-1}N
\] (4.14)

We apply this branch pattern to the two-bit saturating counter defined in Figure 4.3 on page 117 in order to determine an upper bound on the number of mispredictions that can be expected for different initial states of the bimodal branch predictor.

Since the initial state of the counter prior to the execution of the loop construct is usually not known we make the conservative worst-case assumption that it is initially in the *strongly not-taken* (SN) state, which is according to the LT basic pattern defined in Table 4.2 on page 127. Furthermore, we assume that the loop bound \( n \) provided as function parameter is positive such that the branch instruction in basic block \( b_1 \) is *not-taken* because the loop body in basic block \( b_2 \) will be entered at least once.

<table>
<thead>
<tr>
<th>Loop Repetition</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( dir_{res} )</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
</tr>
<tr>
<td>( dir_{pred} )</td>
<td>N</td>
<td>N</td>
<td>T</td>
<td>N</td>
</tr>
<tr>
<td>State (pre)</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>State (post)</td>
<td>01</td>
<td>10</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>Mispredictions</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.3: Predictor behaviour for loop pattern, \( n = 3 \)

Table 4.3 and Table 4.4 on the next page show the behaviour of the branch predictor for a loop with \( n = 3 \) and \( n = 4 \) iterations, respectively. The branch predictor state before (entry state) and after (exit state) is shown for each loop
iteration with the basic invariant that the post state will be the pre state of the following loop iteration.

For the initial loop repetition, the first two iterations of the loop are mispredicted in order to change the prediction from not-taken to taken. Then, the loop itself is repeated a number of times, $m$, until the predictor states repeat themselves, i.e. the predictor is in the weakly taken (WT) state when a new repetition of the loop is entered. In this predictor state, all branches are predicted as taken, which results in a single misprediction for the last iteration of each loop repetition. This last loop iteration changes the predictor state from strongly taken (ST) to weakly taken (WT) such that the first iteration of the next loop repetition is predicted correctly again. It can be observed from the last row in Table 4.3 on the preceding page and Table 4.4 that the number of mispredictions for any repetition of the loop construct is always less than or equal to three. Consequently, for $m$ repetitions of the loop, and each repetition with $n$ iterations, we obtain:

$$\hat{m}p_{\text{loop}}(w) \leq 3m,$$  \hspace{1cm} (4.15)

with $w = (T^{n-1}N)^m$.

\footnote{Note that we distinguish between a loop iteration, i.e. the execution of the loop body in the usual sense, and a loop repetition, which is the execution of the loop construct as a whole. Thus, each loop repetition consists of a number, possibly zero, iterations.}
Table 4.5 summarises the maximum number of branch mispredictions that can be expected in the worst-case for a given number of loop iterations and different initial branch predictor states. The last column of this table also shows the worst-case initial state for a particular number of loop iterations. Note that the behaviour of the bimodal branch predictor is the same for all \( n \geq 4 \). This is because the number of consecutive taken branches for the branch execution pattern \( (T^{n-1}N) \) is at least three and after reaching the strongly taken state any subsequent taken branch does not alter the bimodal predictor state. Finally, the last not-taken branch changes the predictor state to weakly taken.

<table>
<thead>
<tr>
<th>n</th>
<th>Entry State</th>
<th>wc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SN</td>
<td>WN</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.5: Number of mispredictions

We can observe from the results provided in Table 4.5 that the maximum number of mispredictions is given by:

\[
\hat{m}_{p_{\text{loop}}}(w) = \min(n, 3),
\]

with \( w = ( (T^{n-1}N)^m ) \) and where \( n \) is the number of loop iterations.

Let us now assume that the loop construct itself is repeated \( m \) times and the number of loop iterations remains fixed for each repetition of the loop construct itself. The maximum number of mispredictions is now given by:

\[
\hat{m}_{p_{\text{loop}}}(w) = m \cdot \min(n, 3),
\]
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

with \( w = \langle (T^{n-1}N)^m \rangle \) and where \( m \) is the number of times the loop statement is repeated itself and \( n \) is the number of loop iterations.

This figure is pessimistic as it considers each loop repetition in isolation and assumes a worst-case initial predictor state for each repetition of the loop. In the following we will show how the associated pessimism can be reduced to a more acceptable level.

In order to provide a less pessimistic upper bound on the number of branch mispredictions we now consider sequences of loop repetitions and use the predictor exit state of a repetition as the initial predictor state of the subsequent repetition. If the initial bimodal predictor state is unknown, e.g. because the loop is executed for the first time, we assume the worst-case state according to Table 4.5.

Table 4.6 defines the exit states of a bimodal predictor depending on the entry state and the number of loop iterations \( n \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>Entry State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SN SN WN WT</td>
</tr>
<tr>
<td>2</td>
<td>SN WN WT WT</td>
</tr>
<tr>
<td>3</td>
<td>WN WT WT WT</td>
</tr>
<tr>
<td>4</td>
<td>WT WT WT WT</td>
</tr>
</tbody>
</table>

Table 4.6: Predictor exit states

Consider now a loop with \( n = 2 \) iterations. The static branch execution pattern of the loop control branch is \( \langle TN \rangle \), i.e. the branch alternates between the \textit{taken} and \textit{not-taken} path. This pattern represents the worst-case scenario for the bimodal branch predictor and results in a misprediction rate of 100% if the \textit{weakly not-taken} state was the initial state of the predictor. Yet, the upper bound on the number of mispredictions previously stated in Equation (4.15) holds true.
Therefore, for \( n = 2 \) loop iterations and \( m \) repetitions of the loop itself, the upper bound on the number of mispredictions is:

\[
\hat{m}_{p_{\text{loop}}}(w) = 2m, \tag{4.18}
\]

with \( w = \langle (TN)^m \rangle \).

This worst-case scenario does only occur if the initial predictor state is \textit{weakly not-taken}. In all other initial predictor states every second branch is predicted correctly and thus the number of mispredictions per loop repetition is reduced to \( m \), which represent the lower bound on the number of mispredictions.

If we are interested in the number of misprediction for a sequence of loop executions rather than a single execution, we can reduce the pessimism by taking into account the \textit{warm-up} of the branch predictor. Table 4.3 on page 129 indicates that for \( n = 3 \) the predictor states of the third loop repetition repeat themselves for all subsequent repetitions. Thus, the upper bound on the number of branch mispredictions \( m_{p_{\text{loop}}} \) is given by:

\[
\hat{m}_{p_{\text{loop}}}(w) = 3 + 2 + \underbrace{1 + \ldots + 1}_{(m-2)\text{-times}} = 3 + 2 + (m - 2) = 3 + m, \tag{4.19}
\]

with \( w = \langle (T^2N)^m \rangle \) and where \( m \) is the number of times the loop statement is repeated.

As shown in Table 4.4 on page 130, the branch predictor reaches the \textit{strongly taken} state already during the first execution of the loop if \( n \geq 4 \), in other words, the branch is \textit{taken} for at least three subsequent instances. The predictor states repeat themselves already after the second loop repetition. In this case, the upper
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

bound on the number of mispredictions stated in Equation (4.19) changes to:

\[
mp_{\text{loop}}(w) = 3 + 1 + \ldots + 1 \quad \text{(m-1)-times}
\]

\[
= 3 + (m - 1)
\]

\[
= 2 + m,
\]

(4.20)

with \( w = (T^{n-1}N)^m \) and where \( m \) is the number of times the loop statement is repeated and \( n \geq 4 \) is the number of loop iterations.

We determine the upper bound on the number of mispredictions for \( m \) repetitions of the loop construct itself by concatenating the entry/exit states of each individual loop repetition, as illustrated in Table 4.7. In this table, the arrows represent a transition from the worst-case entry state to the corresponding exit state after \( n \) iterations of the loop. The sequence is interrupted once the entry/exit states no longer change. For each transition the number of branch mispredictions according to Table 4.5 on page 131 is given and the last column of the table provides the maximum number of mispredictions for \( m \) repetitions of the loop construct.

<table>
<thead>
<tr>
<th>( n )</th>
<th>Worst-Case Transitions</th>
<th>( mp )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ST ( \rightarrow ) WT ( \rightarrow ) WN ( \rightarrow ) SN ( \rightarrow ) SN</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>WN ( \rightarrow ) WN ( \rightarrow ) WN</td>
<td>( 2m )</td>
</tr>
<tr>
<td>3</td>
<td>SN ( \rightarrow ) WN ( \rightarrow ) WT ( \rightarrow ) WT</td>
<td>( 3 + m )</td>
</tr>
<tr>
<td>( \geq 4 )</td>
<td>SN ( \rightarrow ) WT ( \rightarrow ) WT</td>
<td>( 2 + m )</td>
</tr>
</tbody>
</table>

Table 4.7: Repeated execution of the loop

Theorem 4.1 on the next page summarises our observations and states a bound on the number of mispredictions depending on the number of iterations executed for each loop repetition.

**Theorem 4.1 (Repeated loop)**

*Let \( n \) be the number of loop iterations and \( m \) be the number of times the loop is repeated. Then, the upper bound on the number of mispredictions for the*
repeated execution of a loop statement is defined by:

\[ m_{\text{loop}}(n, m) = \begin{cases} 
2, & \text{if } n = 1 \\
2m, & \text{if } n = 2 \\
3 + m, & \text{if } n = 3 \\
2 + m, & \text{if } n \geq 4 
\end{cases} \]

**Proof:** For \( n = 1 \) the loop exits after its first iteration and therefore the loop control branch is always *not-taken*. The loop control branch is mispredicted only for the first two loop repetitions; any subsequent repetition results in a correct prediction. Hence, the upper bound on the number of mispredictions for \( n = 1 \) is two. The remaining three cases follow immediately from Table 4.7.

\( \square \)

It should be noted that for \( n \geq 3 \) the mispredictions stated in Theorem 4.1 are not equally distributed over the individual iterations of a loop. There are up to three additional mispredictions caused by the initial "warm-up" of the predictor state during the initial two repetitions of the loop. The additional mispredictions mean the first loop repetition always has the highest execution time among all repetitions. The actual number of mispredictions for any loop instance varies between one and three. If we have to guarantee a bound on the number of mispredictions that is valid for all iterations of a loop statement the variation of the execution time due to the initial mispredictions requires us to include a significant amount of pessimism in the WCET analysis. In this case, we have to assume that at most three mispredictions occur for each repetition of the loop construct.

In order to reduce this pessimism the software has to be able to tolerate the variation of execution time during the initial two loop repetitions. These repetitions are used for training the predictor state. Timing requirements have to be only met for subsequent repetitions of the loop. Note that for small loop bodies,
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

Listing 4.2: Loop with embedded conditional construct

```c
int_t loop(int_t *data, int_t n) {
    int_t i, result;

    for (i = 0, result = 0; i < n; i++) {
        if ((data[i] % 2) == 0) {
            data[i] = data[i] / 2;
            result++;
        } else {
            data[i] = data[i] + 1;
        } /* end if */
    } /* end for */

    return result;
} /* end of loop */
```

i.e. the execution time of the loop body is small compared to the misprediction penalty, it is possible that the execution for two loop repetitions can actually be larger than for three loop repetitions due to a higher overall number of branch mispredictions.

4.4.2 Analysis of Conditional Constructs

In our next example we derive an upper bound on the number of mispredictions for a conditional construct. The C source code and the corresponding control flow graph for this example are depicted in Listing 4.2 and Figure 4.6 on the next page, respectively. The condition of the if-then-else statement depends completely on the provided input data, which is usually unknown at compile-time. Thus, it is not possible to determine the outcome of the branch instruction in $b_2$ statically. Before we tackle this problem, let us consider the two execution paths we have to include in the analysis.

Without taking into account the timing effects of branch mispredictions, the WCETs for the two possible execution paths $p_{\text{then}} = \{b_2, b_3\}$ and $p_{\text{else}} = \{b_2, b_4\}$ of the conditional statement in the while-loop are given by the following two
equations:

\[ T(p_{\text{then}}) = T(b_2) + T(b_3) \]  
(4.21)

\[ T(p_{\text{else}}) = T(b_2) + T(b_4) \]  
(4.22)

where \( T(b_i) \) is the WCET of the basic block \( b_i \).

![Control flow graph of a loop with embedded conditional construct](image)

Figure 4.6: Control flow graph of a loop with embedded conditional construct

**Case 1: Not-taken biased branch pattern (BN)**

Let us first consider the simplistic approach used by traditional WCET analysis methods based on the work initially presented by Shaw (1989) that do not take into account the effects of branch prediction or caches. Such methods include only the path with the highest execution time for estimating the WCET of a conditional construct, i.e. for our example:

\[ T(S_{\text{cond}}) = \max \left( T(p_{\text{then}}), T(p_{\text{else}}) \right) \]  
(4.23)

For the loop construct with the embedded conditional construct and \( n \) loop iterations, the WCET estimate is calculated using Equation (4.24), which is based...
on the original timing schema.

\[ T(S_{\text{loop}}) = n \left( T(S_{\text{cond}}) + T(S_{\text{eval}}) \right) \]  

where \( T(S_{\text{eval}}) \) is the WCET estimate of the code fragment representing the evaluation of the loop condition.

In our case, \( S_{\text{eval}} \) corresponds to basic block \( b_5 \), which contains the loop control branch. Thus,

\[ T(S_{\text{eval}}) = T(p_{\text{eval}}) = T(b_5) \]  

Let us now assume the case where the WCET of path \( p_{\text{then}} \) is greater than or equal to the WCET of path \( p_{\text{else}} \):

\[ T(p_{\text{then}}) \geq T(p_{\text{else}}) \]

\[ \Leftrightarrow T(p_{\text{then}}) = T(p_{\text{else}}) + \lambda, \text{ with } \lambda \geq 0 \]  

\[ \Rightarrow T(b_3) = T(b_4) + \lambda \]

According to Equation (4.23) we have to include path \( p_{\text{then}} \) in the calculation of the overall WCET estimate since we assume that \( T(p_{\text{then}}) > T(p_{\text{else}}) \). Therefore, when we repeat the conditional construct for \( n \) iterations of the loop we obtain for its WCET estimate, including now the execution time effects of branch mispredictions:

\[ \tilde{T}(S_{\text{loop}}) = n \left( \max \left( T(p_{\text{then}}), T(p_{\text{else}}) \right) + T(p_{\text{eval}}) \right) + mp \delta \]

\[ = n \left( T(p_{\text{then}}) + T(p_{\text{eval}}) \right) + mp \delta \]  

\[ = n \left( T(p_{\text{else}}) + T(p_{\text{eval}}) + \lambda \right) + mp \delta, \]  

where \( mp \) is the upper bound on the total number of branch mispredictions for the construct \( S_{\text{loop}} \) and \( \delta \) represents the maximum branch misprediction penalty. Substituting Equation (4.26) into Equation (4.28) yields Equation (4.29).
The upper bound on the total number of branch mispredictions, $mp$, in Equations (4.28) and (4.29) combines the number of branch mispredictions due to the conditional construct (branch instruction in basic block $b_2$) and due to the loop construct (branch instruction in basic block $b_5$). Therefore we can write:

$$mp = mp_{\text{cond}} + mp_{\text{loop}}$$  \hfill (4.30)

Bounding the timing effects that occur due to branch mispredictions for Equation (4.28) is straightforward. Assuming *strongly taken* as initial branch predictor state and after executing the conditional construct twice the branch predictor state associated with the branch instruction in basic block $b_2$ is biased toward the *not-taken* direction, i.e. the *weakly not-taken* state, with further loop iterations resulting in the *strongly not-taken* state. The initial mispredictions that occur in the first two iterations of the loop represent a variation of the execution time that is caused by the warm-up of the branch predictor state. All subsequent instances of that branch are predicted correctly. Thus, the maximum number of branch mispredictions for the conditional construct is two. Hence:

$$mp_{\text{cond}} = 2$$  \hfill (4.31)

We can now rewrite Equation (4.28) using $mp = mp_{\text{cond}} + mp_{\text{loop}} = 2 + 3 = 5$ and substitute $T(p_{\text{then}})$ and $T(p_{\text{eval}})$ with Equation (4.21) and Equation (4.25), respectively:

$$\bar{T}_1(S_{\text{loop}}) = n \left( T(b_2) + T(b_3) + T(b_5) \right) + 5\delta$$  \hfill (4.32)

In general, the branch instruction in basic block $b_2$ is biased either toward its *taken* or *not-taken* direction. This branch behaviour corresponds to the BT or BN branch patterns, respectively, defined previously in Table 4.2 on page 127. Therefore, this branch instruction does not exhibit the worst-case behaviour of a bimodal branch predictor. The two worst-case patterns are discussed in the following cases.
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

Case 2: Alternating branch pattern (AN)

The first worst-case scenario in terms of a bimodal branch predictor occurs when the execution of the loop body alternates between the $p_{\text{then}}$ and $p_{\text{else}}$ paths and the bimodal predictor state alternates between the \textit{weakly taken} and the \textit{weakly not-taken} states. In this case, each instance of the branch instruction in basic block $b_2$ results in a branch misprediction.

Let $n$ be the total number of loop iterations. Without loss of generality, we assume that $n$ is even. Then, the overall execution time of the loop construct $S_{\text{loop}}$ for $n$ iterations is given by the following equation:

$$\tilde{T}_2(S_{\text{loop}}) = T(p_{\text{then}}) + T(p_{\text{else}}) + \ldots + n T(p_{\text{eval}}) + mp \delta$$

(4.33)

where $mp$ is the upper bound on the total number of branch mispredictions for the construct $S_{\text{loop}}$ and $\delta$ represents the maximum branch misprediction penalty.

Assuming an alternating branch pattern in Equation (4.33) implies that 100% of the branches in basic block $b_2$ are predicted incorrectly, which is overly pessimistic in most cases as shown by the results of the empirical evaluation in Chapter 3. Repeating the process for $n$ being odd also implies 100% of branches are mispredicted. Therefore, assuming $n$ loop iterations:

$$mp_{\text{cond}} = n$$

(4.34)

We can rewrite Equation (4.33) using $mp = mp_{\text{cond}} + mp_{\text{loop}} = n + 3$ and substitute $T(p_{\text{then}})$, $T(p_{\text{else}})$ and $T(p_{\text{eval}})$ with Equations (4.21), (4.22) and (4.25), respectively. This yields:

$$\tilde{T}_2(S_{\text{loop}}) = n \left( T(b_2) + T(b_5) + \frac{1}{2} T(b_3) + \frac{1}{2} T(b_4) \right) + (n + 3)\delta$$

(4.35)
On the other hand, Equation (4.32) limits the maximum number of branch mispredictions to only two, independent of the number of loop iterations. Consequently, the question arises as to when we have to use $\tilde{T}_1(S_{\text{loop}})$ rather than $\tilde{T}_2(S_{\text{loop}})$ for calculating a WCET estimate.

In order to answer this question we try to find a condition for which the following inequality holds true:

$$\tilde{T}_1(S_{\text{loop}}) \geq \tilde{T}_2(S_{\text{loop}})$$  \hspace{1cm} (4.36)

Basically, we compare the longer execution time of a single path (i.e. the then path in this case) against an alternating sequence of paths with shorter execution time but subject to additional execution time overhead caused by higher number of branch mispredictions.

Substitution of $\tilde{T}_1(S_{\text{loop}})$ and $\tilde{T}_2(S_{\text{loop}})$ in Equation (4.36) yields:

$$n\left(T(b_2) + T(b_3) + T(b_5)\right) + 5\delta \geq n\left(T(b_2) + T(b_5) + \frac{1}{2}T(b_3) + \frac{1}{2}T(b_4)\right) + (n + 3)\delta$$

$$\Leftrightarrow T(b_3) + \frac{2}{n}\delta \geq \frac{1}{2}T(b_3) + \frac{1}{2}T(b_4) + \delta$$

$$\Leftrightarrow T(b_3) \geq 2\delta(1 - \frac{2}{n}) + T(b_4)$$  \hspace{1cm} (4.37)

We use our assumption stated in Equation (4.27) and substitute $T(b_3)$ in Equation (4.37) to obtain:

$$T(b_4) + \lambda \geq 2\delta(1 - \frac{2}{n}) + T(b_4)$$

$$\Leftrightarrow \lambda \geq 2\delta(1 - \frac{2}{n}), \quad \text{with } n > 1$$  \hspace{1cm} (4.38)

Similar considerations apply for the case where $T(p_{\text{else}}) > T(p_{\text{then}})$. Thus,

$$T(p_{\text{else}}) - T(p_{\text{then}}) \geq 2\delta(1 - \frac{2}{n}) \vee T(p_{\text{then}}) - T(p_{\text{else}}) \geq 2\delta(1 - \frac{2}{n})$$  \hspace{1cm} (4.39)

$$\Leftrightarrow |T(p_{\text{then}}) - T(p_{\text{else}})| \geq 2\delta(1 - \frac{2}{n})$$  \hspace{1cm} (4.40)
Note that the condition in Equation (4.40) tends to $2\delta$ for $n \to \infty$. This bound is also conservative, i.e. it is safe to use it instead of the condition given in Equation (4.40) since

$$\forall n > 0 : 2\delta > 2\delta(1 - \frac{2}{n})$$

(4.41)

Thus, using Equation (4.41) we can rewrite Equation (4.40) by making it independent from the number of loop iterations $n$. This yields

$$|T(p_{\text{then}}) - T(p_{\text{else}})| \geq 2\delta$$

(4.42)

Case 3: Alternating branch pattern (ANN)

The second worst-case scenario we consider in terms of a bimodal branch predictor is a variation of the previous case 2 (alternating branch pattern with pattern type AN). It occurs when the execution of the loop body alternates between the $p_{\text{then}}$ and $p_{\text{else}}$ paths, but only after the $p_{\text{then}}$ path, which we assume to exceed the execution time of the $p_{\text{else}}$ path, has been executed twice in succession. In this case, the bimodal predictor state starts in the strongly taken state, changes to the weakly taken state and then alternates between the weakly taken and the weakly not-taken states. Again, each instance of the branch instruction in basic block $b_2$ results in a branch misprediction.

Let $n$ be the total number of loop iterations. According to Table 4.2, the branch pattern type for this case is ANN. The corresponding branch pattern is defined as

$$w = \langle N(NT)^m \rangle, \text{ with } m = \frac{n-1}{2}$$

(4.43)

We assume that $n$ is odd and $m \geq 1$, therefore $n \geq 3$. Note that the previous case covers $n$ being even. Then, the overall execution time, including the effects of branch mispredictions, of the loop construct $S_{\text{loop}}$ for $n$ iterations is given by
the following equation:

\[
\hat{T}_3(S_{\text{loop}}) = 2 T(p_{\text{then}}) + \frac{n-2}{2} (T(p_{\text{then}}) + T(p_{\text{else}})) + n T(p_{\text{eval}}) + mp \delta
\]

(4.44)

where \(mp\) is the upper bound on the total number of branch mispredictions for the construct \(S_{\text{loop}}\) and \(\delta\) represents the maximum branch misprediction penalty.

Note that Equation (4.44) can also be written as:

\[
\hat{T}_3(S_{\text{loop}}) = \hat{T}_2(S_{\text{loop}}) + T(p_{\text{then}}) - T(p_{\text{else}})
\]

(4.45)

\[
= \hat{T}_2(S_{\text{loop}}) + \lambda
\]

(4.46)

Similar to the previous case, considering an alternating branch pattern in Equation (4.44) implies that 100% of the branches instances associated with the conditional statement are mispredicted. Therefore, assuming \(n\) loop iterations provides:

\[
mp_{\text{cond}} = n
\]

(4.47)

Using the same considerations regarding the number of branch misprediction as in the previous case, we can rewrite Equation (4.44):

\[
\hat{T}_3(S_{\text{loop}}) = n \left( T(b_2) + T(b_5) + \frac{1}{2} T(b_3) + \frac{1}{2} T(b_4) \right) + T(b_6) - T(b_4) + (n+3)\delta
\]

(4.48)

As stated earlier for the second case, if the condition in Equation (4.40) is met we have to use \(\hat{T}_1(S_{\text{loop}})\) instead of \(\hat{T}_2(S_{\text{loop}})\) for calculating the overall WCET.
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

estimate of the conditional statement nested in the loop. Otherwise, the calculated WCET estimate may be unsafe. Similarly, we now try to find a condition based on $\hat{T}_3(S_{\text{loop}})$ for which the following inequality holds true:

$$\hat{T}_1(S_{\text{loop}}) \geq \hat{T}_3(S_{\text{loop}})$$

(4.49)

Substitution of $\hat{T}_1(S_{\text{loop}})$ and $\hat{T}_3(S_{\text{loop}})$ in Equation (4.49) yields:

$$T(b_3) + \frac{2}{n}\delta \geq \frac{1}{2}T(b_3) + \frac{1}{2}T(b_4) + \delta + \frac{\lambda}{n}$$

$$\Leftrightarrow T(b_3) - T(b_4) \geq 2\delta(1 - \frac{2}{n}) + \frac{2}{n}\lambda$$

$$\Leftrightarrow \lambda(1 - \frac{2}{n}) \geq 2\delta(1 - \frac{2}{n})$$

$$\Leftrightarrow \lambda \geq 2\delta \land n \geq 3$$

(4.50)

Again, similar considerations apply for the case where $T(p_{\text{else}}) > T(p_{\text{then}})$. Thus,

$$|T(p_{\text{then}}) - T(p_{\text{else}})| \geq 2\delta$$

(4.51)

Finally, we can summarise our findings for the three basic cases of branch execution patterns of a conditional statement within a loop in the following theorem.

**Theorem 4.2 (Repeated conditional statement)**

Let $n$ be the number of times the conditional statement is repeated within the loop body and $\delta$ be the misprediction penalty. Then, the upper bound on the number of branch mispredictions for the repeated execution of a conditional statement is defined by:

$$m_{p_{\text{cond}}}(n, \lambda, \delta) = \begin{cases} 
2, & \text{if } \lambda \geq 2\delta \\
n, & \text{if } \lambda < 2\delta 
\end{cases}$$

with $\lambda = |T(p_{\text{then}}) - T(p_{\text{else}})|$.

**Proof:** $T(p_{\text{then}})$ and $T(p_{\text{else}})$ represent the WCET estimates for the then and else path, respectively. If the condition $\lambda \geq 2\delta$ is met it is safe to restrict the analysis to one of the two possible paths. This path is mispredicted for the first two
iterations of the loop and therefore the maximum number of mispredictions is two. Otherwise, execution alternates between the then and else path and the upper bound on the number of mispredictions follows from Equations (4.33) and (4.44).

\[ \square \]

A similar but less comprehensive approach is presented by Colin and Puaut (2000) but, incorrectly, they use \( \lambda > \delta \) as condition instead. Furthermore, they do not account for the execution time variation that is caused by initial mispredictions until the predictor has reached its steady state. The following example shows that their condition can in fact produce unsafe WCET estimates, i.e. although the condition is met and \( \tilde{T}_1(S_{\text{loop}}) \) is therefore used for the WCET calculation, \( \tilde{T}_1(S_{\text{loop}}) > \tilde{T}_2(S_{\text{loop}}) \) does not always hold true. In order to illustrate this let us assume that \( T(p_{\text{then}}) = 16, \ T(p_{\text{else}}) = 10, \ \text{and } \delta = 5 \) clock cycles. Their condition is clearly met since \( T(p_{\text{then}}) - T(p_{\text{else}}) = 6 > 5 \).

However, for \( n > 5 \) the WCET of the alternating sequence of the two paths is greater than the WCET that only takes into account path \( p_{\text{then}} \), since

\[ \tilde{T}_2(S_{\text{loop}}) > \tilde{T}_1(S_{\text{loop}}) \]

\[ \iff 18n > 16n + 10 \]

\[ \iff n > 5 \]

4.4.3 Multi-Way Decision Statements

Multi-way decision statements can be subdivided into cascaded if-then-else-statements and switch-statements. Listing 4.3 shows the C language source code of a function that contains a cascaded if-then-else-statement embedded into a for-loop. The generated assembler code and the control flow graph of this function are illustrated in Figure 4.7. Basic block \( b_1 \) checks the initial condition of the loop bound whereas basic block \( b_{10} \) contains the loop control branch.
CHAPTER 4. STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS

```
(a) Assembler Code

b1: $004018e8 addiu r29,r29,-8
    $004018f0 addu r3,r0,r0
    $004018f8 addu r6,r0,r0
    $00401900 beq r5,r0,b11

b2: $00401908 addiu r12,r0,1
    $00401910 addiu r11,r0,15
    $00401918 addiu r10,r0,5
    $00401920 addiu r9,r0,20
    $00401928 addiu r8,r0,10
    $00401930 addiu r7,r0,25

b3: $00401938 lw r2,0(r4)
    $00401940 bne r2,r12,b5

b4: $00401948 sw r11,0(r4)
    $00401950 j b10

b5: $00401958 bne r2,r10,b7
    $00401950 j b10

b6: $00401960 sw r9,0(r4)
    $00401968 j b10

b7: $00401970 bne r2,r8,b9
    $00401978 sw r7,0(r4)
    $00401980 j b10

b8: $00401988 addiu r6,r6,1

b10: $00401990 addiu r4,r4,4
    $00401998 addiu r3,r3,1
    $004019a0 sltu r2,r3,r5
    $004019a8 bne r2,r0,b3

b11: $004019b0 addu r2,r0,r6
    $004019b8 addiu r29,r29,8
    $004019c0 jr r31

(b) Control Flow Graph

Figure 4.7: Cascaded conditional construct
```
### 4.4. WCET ANALYSIS OF BASIC CONTROL STATEMENTS

<table>
<thead>
<tr>
<th>Block</th>
<th>Assembler Code</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1: $00402000</td>
<td>addiu r2,r0,400</td>
<td>true</td>
</tr>
<tr>
<td>$00402008</td>
<td>beq r4,r2,b18</td>
<td></td>
</tr>
<tr>
<td>b2: $00402010</td>
<td>slti r2,r4,401</td>
<td>i ≠ 400</td>
</tr>
<tr>
<td>$00402018</td>
<td>beq r2,r0,b9</td>
<td></td>
</tr>
<tr>
<td>b3: $00402020</td>
<td>addiu r2,r0,200</td>
<td>i &lt; 400</td>
</tr>
<tr>
<td>$00402028</td>
<td>beq r4,r2,b16</td>
<td></td>
</tr>
<tr>
<td>b4: $00402030</td>
<td>slti r2,r4,201</td>
<td>i &lt; 400 ∧ i ≠ 200</td>
</tr>
<tr>
<td>$00402038</td>
<td>beq r2,r0,b7</td>
<td></td>
</tr>
<tr>
<td>b5: $00402040</td>
<td>addiu r2,r0,100</td>
<td>i &lt; 200</td>
</tr>
<tr>
<td>$00402048</td>
<td>beq r4,r2,b15</td>
<td></td>
</tr>
<tr>
<td>b6: $00402050</td>
<td>j b22</td>
<td>i &lt; 200 ∧ i ≠ 100</td>
</tr>
<tr>
<td>b7: $00402058</td>
<td>addiu r2,r0,300</td>
<td>i &gt; 200 ∧ i &lt; 400</td>
</tr>
<tr>
<td>$00402068</td>
<td>beq r4,r2,b17</td>
<td></td>
</tr>
<tr>
<td>b8: $00402068</td>
<td>j b22</td>
<td>i &gt; 200 ∧ i &lt; 400 ∧ i ≠ 300</td>
</tr>
<tr>
<td>b9: $00402070</td>
<td>addiu r2,r0,600</td>
<td>i &gt; 400</td>
</tr>
<tr>
<td>$00402078</td>
<td>beq r4,r2,b20</td>
<td></td>
</tr>
<tr>
<td>b10: $00402080</td>
<td>slti r2,r4,601</td>
<td>i &gt; 400 ∧ i ≠ 600</td>
</tr>
<tr>
<td>$00402088</td>
<td>beq r2,r0,b13</td>
<td></td>
</tr>
<tr>
<td>b11: $00402090</td>
<td>addiu r2,r0,500</td>
<td>i &gt; 400 ∧ i &lt; 600</td>
</tr>
<tr>
<td>$00402098</td>
<td>beq r4,r2,b19</td>
<td></td>
</tr>
<tr>
<td>b12: $004020a0</td>
<td>j b22</td>
<td>i &gt; 400 ∧ i &lt; 600 ∧ i ≠ 500</td>
</tr>
<tr>
<td>b13: $004020a8</td>
<td>addiu r2,r0,700</td>
<td>i &gt; 600</td>
</tr>
<tr>
<td>$004020b0</td>
<td>beq r4,r2,b21</td>
<td></td>
</tr>
<tr>
<td>b14: $004020b8</td>
<td>j b22</td>
<td>i &gt; 600 ∧ i ≠ 700</td>
</tr>
<tr>
<td>b15: $004020c0</td>
<td>addiu r2,r0,1</td>
<td>i = 100 → 1</td>
</tr>
<tr>
<td>$004020c8</td>
<td>j b23</td>
<td></td>
</tr>
<tr>
<td>b16: $004020d0</td>
<td>addiu r2,r0,2</td>
<td>i = 200 → 2</td>
</tr>
<tr>
<td>$004020d8</td>
<td>j b23</td>
<td></td>
</tr>
<tr>
<td>b17: $004020e0</td>
<td>addiu r2,r0,4</td>
<td>i = 300 → 4</td>
</tr>
<tr>
<td>$004020e8</td>
<td>j b23</td>
<td></td>
</tr>
<tr>
<td>b18: $004020f0</td>
<td>addiu r2,r0,8</td>
<td>i = 400 → 8</td>
</tr>
<tr>
<td>$004020f8</td>
<td>j b23</td>
<td></td>
</tr>
<tr>
<td>b19: $00402100</td>
<td>addiu r2,r0,16</td>
<td>i = 500 → 16</td>
</tr>
<tr>
<td>$00402108</td>
<td>j b23</td>
<td></td>
</tr>
<tr>
<td>b20: $00402110</td>
<td>addiu r2,r0,32</td>
<td>i = 600 → 32</td>
</tr>
<tr>
<td>$00402118</td>
<td>j b23</td>
<td></td>
</tr>
<tr>
<td>b21: $00402120</td>
<td>addiu r2,r0,64</td>
<td>i = 700 → 64</td>
</tr>
<tr>
<td>$00402128</td>
<td>j b23</td>
<td></td>
</tr>
<tr>
<td>b22: $00402130</td>
<td>addiu r2,r0,0</td>
<td>others → 0</td>
</tr>
<tr>
<td>b23: $00402138</td>
<td>jr r31</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.8: A switch-statement translated into assembler code
Listing 4.3: Cascaded conditional construct

```c
int_t tc_filter(int_t *data, int_t n)
{
    int i;
    int not_found = 0;

    for(i = 0; i < n; i++) {
        if (data[i] == 1) {
            data[i] = 15;
        } else if (data[i] == 5) {
            data[i] = 20;
        } else if (data[i] == 10) {
            data[i] = 25;
        } else {
            not_found++;
        } /* end if */
    } /* end for */

    return not_found;
} /* end of tc_filter */
```

A `switch`-statement represents another form of a multi-way decision statement. It can only be used in certain cases where, first, the decision depends on a single variable that is of ordinal type, and, second, each possible value of the condition variable can control a single path, i.e. the condition variable is not checked for value ranges. The assembler code generated for a `switch`-statement depends upon the number and distribution of the individual case values. If the case values are distributed over a small range, the compiler generates a jump target table containing the addresses of code locations to be executed for different cases. At run-time, the value of the condition variable is translated into an index into the jump table and the indexed target address is stored in a register that is used by a register indirect jump instruction to transfer control to the appropriate code location. The advantage of this approach is that only one jump instruction is necessary independent of the actual number of case values. However, the indirect jump instruction experiences an address misprediction each time a different target address is used from the jump table. As far as static analysis for branch predictors is concerned, we have to assume that an address misprediction occurs...
for each execution of the `switch`-statement.

A different implementation approach has to be used when the case values are distributed over a wide range so that the jump table would become too large. In this case, the compiler uses a sequence of conditional branch instructions to transfer control to the appropriate code location. This approach is illustrated by Figure 4.8, which shows the assembler code generated for a `switch`-statement with seven case blocks and one default block. The assembler code is annotated with preconditions for each basic block, where \( i \) denotes the value of the condition variable. Compared with the cascaded conditional statement discussed earlier the control flow is now much more complex. The maximum number of conditional branches required to resolve a case condition is five instead of seven for the cascaded conditional statement. For this implementation approach, we have to assume that at most five branch mispredictions occur for each execution of the `switch`-statement. It is not possible, however, to state a general bound on the number of mispredictions depending on the number of cases because the distribution of the case values has also an impact on the generation of the control flow graph.

### 4.5 Case Study

The purpose of this section is to illustrate the use of the analysis models presented and discussed earlier in this chapter through a case study. We will evaluate in this case study the behaviour of a code example written in the C programming language using three different execution scenarios.

Listing 4.4 on page 151 shows the C source code for the NP-A sample function, which contains a conditional statement embedded within a loop construct. The `condloop` sample program used for the evaluation of conditional statements consists of a set of functions containing `for`-loop constructs with embedded
if-then-else statements. Each of these functions is a variation of the one shown in Listing 4.4 on the facing page. The execution time of the not-taken paths, which corresponds to the if path in the source code, exceeds that of the taken path for the NP-A and NP-B sample functions. The NP-B sample function, as shown in Listing C.2 on page 247, contains one additional statement in its then path in order to further increase the execution time difference of the two paths. This additional statement corresponds to eight additional assembler instructions. For the TP-A and TP-B sample functions the taken path, which corresponds to the else path, has the longest execution time. This is achieved by negating the condition and swapping the two paths of the if-then-else statement of the original NP-A sample function shown in Listing 4.4 on the facing page.

Figure 4.9 on page 152 provides the MIPS PISA assembler code generated by the GCC cross-compiler for the NP-A sample function and the corresponding control flow graph, which is identical for all four sample functions – only the length and ordering of the basic blocks $b_3$ and $b_4$ shown here are changed. The loop construct of our sample function iterates over an array data with num_samples sample values and the condition of the if-then-else statement checks for each value data[i] whether it is either even or odd. We assume that the number of sample values is 100. Therefore, the loop body is executed num_samples times. For the NP-A sample function shown in Listing 4.4, the taken path is executed when data[i] contains an even value.

According to the control flow graph depicted in Figure 4.9(b) on page 152, there are three conditional branch instructions located in basic blocks $b_1$, $b_2$, and $b_5$. In addition, basic block $b_3$ contains an unconditional branch instruction, for which we assume that the BTB mispredicts the target address once on the first instance of the branch and the target address of all subsequent instances is then found in the BTB. The branch instruction in basic block $b_5$ evaluates the loop condition at the bottom of the loop construct, whereas, in addition, the branch
Listing 4.4: Conditional construct embedded within loop – NPA

```c
typedef struct result_struct {
    ulong_t num_even;
    ulong_t sum;
    ulong_t qsum;
    ulong_t csum;
} result_t;

result_t cond_loop_npa(ulong_t *data,
                        ulong_t *out,
                        ulong_t num_samples)
{
    result_t res;
    ulong_t i;

    res.num_even = 0;
    res.sum = 0;
    res.qsum = 0;

    for (i=0; i<num_samples; i++) {
        if ((data[i] % 2) == 0) {
            /* even value -> not-taken path (basic block b3) */
            out[i] = data[i] / 2;
            res.sum += data[i];
            res.qsum += data[i] * data[i];
            res.num_even++;
        } else {
            /* odd value -> taken path (basic block b4) */
            out[i]++;
        } /* end if */
    } /* end for */

    return res;
}
```

151
CHAPTER 4. STATICAL ANALYSIS OF BIMODAL BRANCH PREDICTORS

(a) Assembler Code

(b) Control Flow Graph

Figure 4.9: Conditional statement embedded within loop
in $b_1$ initially evaluates the loop condition prior to entering the loop body for the first time. This additional basic block is introduced by the code transformation due to the loop inversion optimisation technique.

The branch instruction in basic block $b_1$ needs to be present because in our example program the number of loop iterations is provided as a variable (i.e function parameter num_samples), which may be set to zero by the calling function – meaning that the loop body will not be executed at all. In order to cover this case, the compiler has to generate an additional test at the loop entry. This additional basic block introduced into the generated code is good example of a node in the control flow graph not immediately obvious from the C source code.

Finally, basic block $b_2$ contains the branch instruction that is associated with the condition of the if-then-else statement. The two possible execution paths of the loop construct due to this conditional statement are:

$$p_{\text{then}} = \{b_2, b_3\}$$

$$p_{\text{else}} = \{b_2, b_4\}$$

### 4.5.1 Execution Scenarios

For the purpose of this evaluation, we distinguish between the following three execution scenarios:

1. *always not-taken path*; assuming that the execution time of the *not-taken* path exceeds that of the *taken* path, this scenario represents the WCET in terms of the original timing schema proposed by Shaw (1989), i.e. ignoring the effects of branch mispredictions on overall execution time. The taken rate is 0%.

2. *alternating paths*; depending on the initial state of the branch predictor this scenario exhibits the worst-case predictor behaviour, i.e. all branch
instances are mispredicted. For this scenario, the taken rate is 50%.

3. *always taken path*; this scenario represents the best-case both in execution time and number of branch mispredictions. In contrast to the other two scenarios, we assume for all branch instructions an initial predictor state that causes best-case predictor behaviour. The taken rate for this scenario is 100%.

Table 4.8 shows, for the three execution scenarios described above, the number of times $d_{ij}$ each transition $b_i \rightarrow b_j$ in the control flow graph is executed and the corresponding number of branch mispredictions $mp_{ij}$. We set $mp_{ij} = 0$ if $d_{ij} = 0$. Without loss of generality, we assume that the total number of loop iterations $n$ is even. Note that basic block $b_4$ is the only one in the control flow graph that does not contain a branch instruction and therefore does not contribute to the number of branch mispredictions.

<table>
<thead>
<tr>
<th>$b_i \rightarrow b_j$</th>
<th>Not-taken</th>
<th>Taken</th>
<th>Alternating</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$d_{ij}$</td>
<td>$mp_{ij}$</td>
<td>$d_{ij}$</td>
</tr>
<tr>
<td>$b_1 \rightarrow b_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$b_1 \rightarrow b_6$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$b_2 \rightarrow b_3$</td>
<td>$n$</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$b_2 \rightarrow b_4$</td>
<td>0</td>
<td>0</td>
<td>$n$</td>
</tr>
<tr>
<td>$b_3 \rightarrow b_5$</td>
<td>$n$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$b_4 \rightarrow b_5$</td>
<td>0</td>
<td>0</td>
<td>$n$</td>
</tr>
<tr>
<td>$b_5 \rightarrow b_2$</td>
<td>$n-1$</td>
<td>2</td>
<td>$n-1$</td>
</tr>
<tr>
<td>$b_5 \rightarrow b_6$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total $mp$</strong></td>
<td><strong>7</strong></td>
<td><strong>1</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.8: Number of mispredictions for a conditional construct
While the number of mispredictions is constant for both the always *not-taken* and always *taken* path cases, it is proportional to the number of loop iterations for the alternating paths case. In particular, it is important to note that the worst-case scenario in terms of bimodal branch prediction and the scenario representing the worst-case when considering the execution time of the two paths in isolation are *mutually exclusive* in practice. Consequently, combining these two worst-case scenarios will result in overly pessimistic WCET estimates.

### 4.5.2 Comparing the Execution Times of the Scenarios

Table 4.9 shows the execution time in clock cycles for the four sample loops assuming 100 loop iterations each and a perfect branch predictor, i.e. all branch instructions are predicted correctly by the predictor. The columns titled *Ratio* give the execution time ratio relative to the execution time of the NP-A sample loop when executing the always *taken* scenario. The last column in the table provides the average execution time difference per loop iteration between the always *not-taken* and always *taken* scenario, which is defined as:

\[ \lambda = |T(p_{\text{then}}) - T(p_{\text{else}})| \]  

<table>
<thead>
<tr>
<th>Sample</th>
<th>Not-taken</th>
<th>Alternating</th>
<th>Taken</th>
<th>( \lambda )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycles</td>
<td>Ratio</td>
<td>Cycles</td>
<td>Ratio</td>
</tr>
<tr>
<td>NP-A</td>
<td>3028</td>
<td>2.4658</td>
<td>2128</td>
<td>1.7329</td>
</tr>
<tr>
<td>NP-B</td>
<td>4330</td>
<td>3.5251</td>
<td>2780</td>
<td>2.2638</td>
</tr>
<tr>
<td>TP-A</td>
<td>1328</td>
<td>1.0814</td>
<td>2128</td>
<td>1.7329</td>
</tr>
<tr>
<td>TP-B</td>
<td>1330</td>
<td>1.0831</td>
<td>2780</td>
<td>2.2638</td>
</tr>
</tbody>
</table>

Table 4.9: Execution time for a conditional construct assuming perfect predictor
Although we assume a perfect branch predictor in this example it should be noted that the instruction pipeline is still subject to stalls due to pipeline hazards, for example, dependencies among instructions (data hazards) and limited processor resources (structural hazards). In particular, the simulated perfect branch predictor only assumes a zero clock cycle branch misprediction penalty and does not eliminate control hazards, which interrupt the smooth flow of instructions being fetched. This means that the degree of instruction overlap within the pipeline and the number of instructions that can be issued simultaneously is lower than possible theoretically (i.e. as derived from instruction pipeline design). As we will see in Chapter 5 this explains the fact that for our three execution scenarios the execution times recorded in this table exceed that of a realistic branch predictor, for example, a bimodal predictor.

Let us now assume the case where the WCET of path $p_{\text{then}}$, i.e. the not-taken path of the branch instruction in basic block $b_2$, is greater than or equal to the WCET of path $p_{\text{else}}$:

$$T(p_{\text{then}}) = T(p_{\text{else}}) + \lambda, \quad \text{with } \lambda \geq 0$$

$$\Leftrightarrow T(b_3) = T(b_4) + \lambda \quad (4.53)$$

In Table 4.9 on the preceding page, the case assumed in Equation (4.53) corresponds to the NP-A and NP-B sample functions, which we will primarily consider in the following discussion. In general, the interpretation of the evaluation results is similar for the TP-A and TP-B sample functions – just the taken and not-taken cases are interchanged.

It is interesting to note, however, that the execution time difference $\lambda$ shown in Table 4.9 for these two sample functions is smaller than for the NP-A and NP-B sample functions. Although the paths in the if-then-else statement are simple interchanged this results in slightly different overlapping of instructions between basic blocks $b_2-b_3$ and $b_2-b_4$, respectively, in the pipeline.
4.5.3 Evaluation

Figure 4.10 shows the estimated execution time for the three execution scenarios taking into account the number of branch mispredictions $mp$ defined in Table 4.8 on page 154.

Figure 4.10: Estimated execution time of a conditional statement

The estimated execution time figures for each scenario and sample function have been calculated by adding the total misprediction penalty $mp \cdot \delta$ to the
corresponding measured execution times provided in Table 4.9 on page 155. For each of the four sample functions, the plots in Figure 4.10 on the preceding page show the execution times in clock cycles for a perfect branch predictor (zero cycle branch misprediction penalty) as shown in Table 4.9 on page 155 and also assuming a branch misprediction penalty (mpp) of three and 15 clock cycles, respectively. As mentioned earlier in this section, for the NP-A and NP-B sample functions the always taken execution scenario assume best-case branch predictor behaviour while the always not-taken and alternating scenarios assume worst-case predictor behaviour.

We can observe from Figure 4.10 on the preceding page that when assuming a misprediction penalty of three clock cycles, the always not-taken scenario (0% taken rate) represents the WCET for both the NP-A and NP-B sample functions. This changes, however, when we increase the misprediction penalty to 15 clock cycles. In this case, the WCET for the NP-A sample function is now represented by the alternating paths scenario (50% taken rate). The reason is that the execution time including the additional clock cycles introduced by each mispredicted branch instruction now exceeds the execution time of the case where only the longest path is executed.

4.6 Branch Interference

We have not assumed a particular implementation of a bimodal branch predictor so far in this chapter. In practice, however, the number of entries in the BHT of a bimodal predictor is of limited size, so different branch instructions may have to share the same two-bit counter state if their instruction addresses map to the same entry in the BHT. This effect is called branch interference or branch aliasing. According to Young et al. (1995), we can distinguish between three different types of branch interference:
4.6. BRANCH INTERFERENCE

- A branch interference is classified as constructive if the branch outcome is predicted correctly and an interference-free predictor mispredicts the outcome;

- Conversely, if the outcome is mispredicted but the interference-free predictor predicts it correctly we classify the interference as destructive;

- Otherwise, the interference is classified as neutral.

The event of interference occurs when a branch instruction uses a predictor state entry that has been previously updated by a different branch instruction. Interfering branches simply reuse the history of the other branches sharing the same entry in the predictor state. This is in contrast to conflict misses in instruction and data caches, which are always destructive and require the replacement of the affected cache entry.

4.6.1 Simulating the Effects of Branch Interference

Table 4.10 on the next page shows the simulation results for the bzip2 sample program using various configurations of a bimodal branch predictor. The first column in this table gives the number of bits that are used to index the branch history table (BHT). Note that a BHT size of 12 bits corresponds to the simulation results reported earlier in Chapter 3 (see also Table 3.8 on page 91).

The total number of branch mispredictions for each BHT configuration is provided in the second column while the third column shows the average branch misprediction rate. We can conclude from the results that the branch prediction accuracy has the tendency to improve with increasing size of the BHT. This is partly due to the interference between different branch instructions when they access the same entry in the BHT. The results of the branch interference analysis is
## Table 4.10: Interference effects for bimodal predictor (bzip2)

<table>
<thead>
<tr>
<th>BHT</th>
<th>Mispredictions</th>
<th>Interference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Ratio</td>
</tr>
<tr>
<td>4</td>
<td>3,642,426</td>
<td>7.74%</td>
</tr>
<tr>
<td>5</td>
<td>3,699,827</td>
<td>7.86%</td>
</tr>
<tr>
<td>6</td>
<td>3,225,270</td>
<td>6.85%</td>
</tr>
<tr>
<td>7</td>
<td>3,202,148</td>
<td>6.80%</td>
</tr>
<tr>
<td>8</td>
<td>3,171,469</td>
<td>6.74%</td>
</tr>
<tr>
<td>9</td>
<td>3,118,758</td>
<td>6.62%</td>
</tr>
<tr>
<td>10</td>
<td>3,117,690</td>
<td>6.62%</td>
</tr>
<tr>
<td>11</td>
<td>3,095,942</td>
<td>6.58%</td>
</tr>
<tr>
<td>12</td>
<td>3,095,810</td>
<td>6.58%</td>
</tr>
<tr>
<td>13</td>
<td>3,105,894</td>
<td>6.60%</td>
</tr>
<tr>
<td>14</td>
<td>3,105,883</td>
<td>6.60%</td>
</tr>
<tr>
<td>15</td>
<td>3,105,878</td>
<td>6.60%</td>
</tr>
<tr>
<td>ideal</td>
<td>3,105,878</td>
<td>6.60%</td>
</tr>
</tbody>
</table>

CHAPTER 4. **STATIC ANALYSIS OF BIMODAL BRANCH PREDICTORS**
provided in the remaining columns, which denote the number of mispredictions associated with constructive, destructive, neutral, and no interference, respectively. These results are obtained by comparing the predictions of each simulated BHT configuration with an ideal (interference-free) bimodal predictor, which uses a fully associative BHT. For a BHT size of 15 bits, no branch interference occurs any more and therefore the mispredictions results are the same as for an ideal bimodal branch predictor.

It can be observed from the interference analysis results that an ideal predictor does not necessarily perform better than a non-ideal predictor. According to the results provided in Table 4.10 on the preceding page, a BHT size of 12 bits leads to the lowest number of branch mispredictions for bzip2 sample program. This is the case when constructive interference dominates over destructive interference or when the branch history captured in the BHT is modified due to interference such that it reflects the true behaviour of the programs more accurately. The table does not address the latter case because the branch interference analysis performed for this example only tracks branches that access a BHT entry after another branch has previously accessed the same entry, i.e. the actual interference case. However, constructive or destructive interference alters the history recorded in the BHT and subsequent branch instances may benefit from this even though no additional interference occurred. Interference between branches leads to a correlation of their branch histories stored in the BHT entry they share. As pointed out earlier, for some branches the prediction accuracy benefits from such correlation.

Branch interference requires that we widen the scope of the static analysis to cover global effects across multiple branch instructions in order to identify all branch instructions that may suffer from destructive interference and, therefore, actually cause additional mispredictions.
4.6.2 Impacts of Branch Interference on Static Analysis

As far as static WCET analysis is concerned, we need to address the effects of interference among branches because this may invalidate any assumption regarding the behaviour of the involved branches when analysed separately. Branch interference complicates static analysis because we have to widen the scope of the analysis such that the execution behaviour of multiple branch instructions can be taken into account. It is tempting to believe that addressing any occurrence of destructive interference, in the sense of the original definition provided by Young et al. (1995), is sufficient and constructive interference can be completely ignored as it will only increase the pessimism associated with our static analysis approach. However, this is not necessarily the case – as we will show in the following.

In order to illustrate the effects of branch interference on predictor behaviour, we consider two branch instructions $b_1$ and $b_2$ whose execution behaviour is given by the strings $w_1, w_2 \in \Sigma^*$. We assume that both branch instructions interfere with each other because they are mapped to the same entry in the BHT of a bimodal branch predictor, i.e. $\text{lookup}(b_1) = \text{lookup}(b_2)$. Let us further assume that the branches are embedded within a loop construct and therefore are repeated in sequence several times. The branch instruction associated with the loop construct does not interfere with the other two branches. We will distinguish between two scenarios regarding the interference between the two involved branches:

1. The instances of the first branch instruction, represented by the string $w_1$, are executed and then followed by the instances of the second branch, and so on. This is the case when the two branches are associated with or located within loop constructs. The effect of branch interference is that the execution of the first branch may alter the initial predictor state of the second branch and vice versa. The actual resulting execution pattern that
is seen by the predictor is simply the alternating concatenation of the two strings \( w_1 \) and \( w_2 \).

2. The interference occurs between single instances of the two branch instructions involved. In this case, the branches are typically associated with conditional statements. The string that represents the behaviour of the branch predictor is now generated by interleaving of the original two strings. In particular, the strings \( w_1 \) and \( w_2 \) are no longer substrings of the new execution pattern. Yet, we are able to determine the resulting string if the behaviour of the two branches is known statically. If we assume, for example, that the first branch is \textit{taken} biased and the second is \textit{not-taken} biased then interference between these two branches will cause an alternating execution pattern, which represents worst-case predictor behaviour.

4.6.3 **Addressing the Branch Interference Problem**

In the context of WCET analysis, the problem of interference among branch instructions can be tackled from two different sides:

- Model the effect of branch interference in the static WCET analysis approach. This is the most desirable option but it also requires a more complex model and therefore complicates the analysis. For the approach presented earlier in this chapter, for example, this would mean that the assumptions regarding the initial predictor states are no longer necessarily valid in the case of branch interference.

- Avoid branch interference by changing the instruction address of affected branch instructions, for example, by introducing \texttt{nop} instructions into the assembler code. Unfortunately, this requires an algorithm with polynomial execution time. Although this approach may not be feasible for all cases
where branches interfere with each other, it may be beneficial – also from a performance point of view – in cases where the branch interference is destructive and therefore leads to worst-case predictor behaviour. Zhao et al. (2004) discuss the repositioning of complete basic blocks in order to reduce the WCET estimate.

We propose to use the latter approach as it does not further complicate the complexity of the branch prediction analysis model itself, the first approach would be difficult to analyse without significant pessimism and it can also help to reduce the WCET in cases of destructive branch interference.

In order to keep the number of branch instructions that have to be relocated as small as possible we extend our original branch classification model presented in Section 4.3 on page 120 to identify branch instructions subject to destructive interference as follows:

- **Hard-to-predict** branch instructions are not relocated because these branches are already assumed to exhibit worst-case branch predictor behaviour.

- **Easy-to-predict** branch instructions, in contrast, are further classified into those being biased toward the *taken* direction and those being biased toward the *not-taken* direction. Then, all branch instructions that both share the same entry in the BHT and are biased to different branch directions are required to be relocated. The number of branch instructions to be relocated can be further reduced, if necessary, by not considering interference among branches that are on mutually exclusive execution paths.

### 4.7 Summary

This chapter has presented a static analysis method to calculate an upper bound on the number of branch mispredictions for bimodal branch predictors. We have
extended the classification approach previously used for data cache analysis to
address dynamic branch predictors by distinguishing between branch instructions
that are easy-to-predict and branch instructions that are hard-to-predict. The
distinction is that the behaviour of easy-to-predict branches can be determined
from their semantic context at compile-time, whereas hard-to-predict branches
have to be excluded from analysis and the pessimism caused by these branches has
to be accepted. Using this classification approach, an upper bound on the number
of branch mispredictions has been derived for various control statements of the
C programming language, like loop constructs and conditional statements. For
the latter, the condition originally stated by Colin and Puaut (1999) has been
corrected. The benefit of using the static analysis method has been presented
through a case study.
Chapter 5

Integration with Pipeline Analysis

In this chapter, we discuss how the static analysis method for bimodal branch predictors presented in the previous chapter can be integrated with instruction pipeline analysis in order to estimate the WCET using an ILP-based calculation method. The results of branch prediction analysis are used to define additional ILP constraints on the execution counts of transitions between basic blocks.

In this chapter, Section 5.1 provides an overview of the principles of instruction execution using pipelined microprocessor architectures. Instruction pipelining supports the parallel execution of instructions in a microprocessor – also referred to as instruction-level parallelism (ILP). The instruction pipeline configuration used as reference in this chapter is based on the implementation in the SimpleScalar microprocessor simulator. Section 5.2 defines the instruction pipeline model that is based on a technique previously published by Engblom et al. (1999) and Engblom and Ermedahl (1999) and extends this model in order to include the pipeline effects of branch mispredictions. Section 5.3 illustrates the integration of static WCET analysis for bimodal branch predictors with the instruction pipeline model using a case study. The WCET estimates are calculated using a tree-based approach. Section 5.4 introduces the concept of program flow analysis based on Implicit Path Enumeration Technique (IPET), which mod-
els the control flow of a program as a sequence of constraints on the execution count variables for each basic block in the control flow graph. The resulting cost function is solved using Integer Linear Programming (ILP) techniques, which is demonstrated in Section 5.5 for our case study. Finally, Section 5.6 summarises the results presented in this chapter.

5.1 Overview of Instruction Pipelining

Early microprocessors strictly followed the von Neumann architecture principle of instruction sequentiality. Instructions were issued and executed one after another. Figure 5.1(a) shows the sequential, non-pipelined execution of a two instruction sequence. In order to improve processing performance, modern microprocessors exploit micro-architectural features that make use of the implicit parallelism inherent in programs.

![Sequential (non-pipelined)](IF ID EX MEM WB IF ID EX MEM WB)

(a) Sequential (non-pipelined)

![Overlapping (pipelined)](IF ID EX MEM WB IF ID EX MEM WB IF ID EX MEM WB IF ID EX MEM WB IF ID EX MEM WB IF ID EX MEM WB)

(b) Overlapping (pipelined)

Figure 5.1: Principles of instruction execution

5.1.1 Basic Concepts

The parallel execution of instructions in a microprocessor is called instruction-level parallelism (ILP). In contrast to parallelism among tasks, which is also
5.1. OVERVIEW OF INSTRUCTION PIPELINING

referred to as \textit{process-level parallelism}, instruction-level parallelism remains transparent to both the compiler and the assembler programmer. A key implementation technique to exploit instruction-level parallelism is \textit{instruction pipelining}. This is the overlapping execution of multiple instructions by splitting them into several steps that operate separately. Each step represents a stage of the instruction pipeline. The principle of overlapping execution of an instruction sequence on a five-stage microprocessor pipeline is depicted in Figure 5.1(b). Although contemporary microprocessors usually use more pipeline stages this architectures suffices to illustrate the underlying principle of pipelining. The pipelining stages shown in the figure correspond to a microprocessor data-path split into the following five processing steps:

- \textit{Instruction fetch} (IF). Fetches the instruction referenced by the program pointer (PC) from memory.

- \textit{Instruction decode} (ID). Reads the processor registers and decodes the instruction.

- \textit{Execution} (EX). Executes the operation or calculates an address.

- \textit{Memory access} (MEM). Operands in data memory are accessed.

- \textit{Write back} (WB). This stage stores the result of the instruction execution into the register file.

It is important to note that instruction-level parallelism does not reduce the time required to execute a single instruction, but rather improves the overall instruction throughput of the microprocessor and therefore lowers the program execution time as a whole (Hennessy and Patterson, 2006).
5.1.2 Pipeline Hazards

Each pipeline stage takes a single clock cycle and thus overlapping execution on a single pipeline allows the fetch unit to issue one instruction per cycle. In practice, however, smooth overlapping of the execution of instructions is not always possible. Situations may occur where the next instruction in a pipeline cannot execute in its designated clock cycle due to dependencies between instructions or pipeline resource conflicts (for example, an instruction provides a result that is required by a later instruction). Such events are called pipeline hazards, or pipeline stalls, and are basically an aspect of the instruction pipeline design of the microprocessor.

Three different types of hazards could occur in the pipeline:

- **Structural hazards** arise from resource conflicts due to limited processor resources. For example, in a processor with a single floating point unit, two instructions may want to use that unit at the same time.

- **Control hazards** arise from instructions that change control flow, such as jump and branch instructions. This type of hazard can significantly diminish the performance of the instruction pipeline when a branch is taken. In this case, the instructions issued after the branch have to be removed (flushed) from the pipeline and the branch target must be fetched instead.

Figure 5.2: Example of a pipeline stall

Figure 5.2 shows an example of a pipeline hazard due to a data dependency between the second instruction and the last two instructions in the pipeline.
Exceptions are another form of control hazards. In this case, the problem is to associate an exception with the correct instruction in the pipeline.

- **Data hazards** arise from overlapping instructions that would change their access order to an operand (true data dependence), or from using the same storage location (name dependence).

### 5.1.3 Instruction Issue Policy

There exist two principles on how to proceed with pipeline operation in the case where a pipeline hazard has occurred.

- **In-order instruction execution** continues execution of the instruction causing the stall and of all instructions issued earlier to the pipeline, but interlocks all later instructions until the dependency or conflict is resolved. Also, no new instructions are fetched or issued to the pipeline. All instructions complete in program order because the original instruction sequence is not changed. The disadvantage of this approach is that it only utilises a small degree of parallelism among instructions and thus the instruction throughput of the pipeline is considerably degraded.

- **Out-of-order instruction execution** issues an instruction to the execution stage of the pipeline as soon as there is no unresolved dependency with an earlier instruction and there is a functional unit available. Thus, the sequence of instructions executed in the pipeline is not necessarily in the same order as in the original assembler code (program order) and instructions may complete out-of-order. In order to satisfy the von Neumann architecture principle, i.e. the sequential order of results, the completed instructions have to be rearranged using a reorder buffer mechanism. After instructions are completed, the reorder buffer ensures that their results are written-back (retired) in the original program order they were fetched.
The instruction issue rate of a processor is indicated by the measure instructions per clock cycle (IPC). Together with the measure clock cycles per instruction (CPI), which is the inverse of IPC, these two measures are commonly used to compare different instruction-set architectures and implementation techniques.

The number of clock cycles required to execute \( n \) instructions on a single-issue instruction pipeline with \( k \) stages is \( k + n - 1 \), assuming that no pipeline stalls occur, for example, due to mispredicted branches. The CPI in this case is given by

\[
CPI_{\text{scalar}} = \frac{n + k - 1}{n} = 1 + \frac{k - 1}{n} \quad (5.1)
\]

Hence, for a single-issue microprocessor the ideal CPI, and also the IPC, are equal to one if the number of instructions issued to the pipeline tends to infinity. Pipeline stalls limit the degree of overlapping between subsequent instructions and thus additional cycles are required. The worst-case scenario for an instruction pipeline is when instructions do not overlap at all, that is, each instruction is issued only after the preceding one has been completed. In this case, the execution of \( n \) instructions requires \( n \times k \) clock cycles and the CPI for strictly sequential instruction execution is given by

\[
CPI_{\text{seq}} = \frac{n \times k}{n} = k \quad (5.2)
\]

The theoretical performance enhancement of a single-issue instruction pipeline for an infinite number of instructions is

\[
\text{Speedup} = \frac{CPI_{\text{seq}}}{CPI_{\text{scalar}}} = k \quad (5.3)
\]

### 5.2 Modelling Instruction Pipelines

A simplified version of the instruction pipeline timing analysis technique described by Engblom et al. (1999) and Engblom and Ermedahl (1999) is used. Their technique addresses the retargetability problem of pipeline timing analysis by using
a trace-driven microprocessor simulator instead of pipeline modelling to determine the execution time effect of pipeline overlap between successive instructions and basic blocks. In Engblom’s work, the analysis deals with cases where the pipeline overlap can span multiple blocks. An example of how this can occur are some floating-point instructions that are long-running instructions and take many clock cycles to execute. Engblom’s original approach is simplified in the sense that it is assumed capturing only the effects of pipeline overlap across pairs of basic blocks is sufficient. The reason being is that Engblom’s method increases the complexity of the timing analysis which could obscure from the focus of the following discussion.

5.2.1 Overlap Between Basic Blocks

In general, the amount of overlap, $\delta_{ij}$, across two adjacent basic blocks $b_i$ and $b_j$ is calculated as follows:

$$\delta_{ij} = t_{ij} - (t_i + t_j),$$

(5.4)

with $\forall i, j \cdot (b_i \rightarrow b_j) \in E,$

where $t_{ij}$ is the number of clock cycles required to execute $b_i$ and $b_j$ in succession; $t_i$ and $t_j$ are the number of clock cycles required for executing $b_i$ and $b_j$, respectively, in isolation.

We introduce a special vertex $\pi$ in order to define a predecessor for the first executed basic block and set $t_{\pi 1} = t_1$ and $t_{\pi} = 0$. This will simplify the formal expression of the WCET calculation formula later in this section.

5.2.2 Including the Effects of Mispredictions

A misprediction of the branch instruction in basic block $b_i$ reduces the amount of pipeline overlap between basic block $b_i$ and the block $b_j$ on its correct branch.
target. This may even result in a positive value for $\delta_{ij}$, which would occur when there is no pipeline overlap but instead a pipeline stall. Also, it may have an impact on the overlap among instructions within basic block $b_j$ and therefore change the execution time of $b_j$. Although Equation (5.4) implicitly takes into account a misprediction of the branch instruction in basic block $b_i$ we distinguish between the mispredicted and correctly predicted case in order to be able to provide a more accurate WCET estimate, which will be more evident later in this chapter. Thus, we calculate the pipeline overlap, $\delta_{ij}^{mp}$, in the case of a branch misprediction according to the following equation:

$$\delta_{ij}^{mp} = t_{ij}^{mp} - (t_i + t_j^{mp}),$$  \hspace{1cm} (5.5)

with $\forall i, j \cdot (b_i \rightarrow b_j) \in E \land b_i \in V_c$,

where $t_{ij}^{mp}$ is the maximum number of clock cycles required to execute $b_i$ and $b_j$ in succession, and $t_j^{mp}$ is the maximum number of clock cycles required for executing $b_j$ taking into account the effects of a misprediction of the branch instruction contained in $b_i$.

### 5.3 Case Study: Tree-Based WCET Analysis

In this section, we discuss a brief example that shows how the static analysis method for bimodal branch predictors presented in Chapter 4 can be integrated with pipeline analysis.

We use four sets of sample data to generate different execution scenarios. For our example, an execution scenario consists of an execution sequence of basic blocks together with branch misprediction events and the values of the instruction operands. The selected sets of sample data are sufficient to generate all combinations of branch misprediction events. We use the sim-outorder cycle-level simulator to execute each execution scenario in order to obtain the execution
time, which is measured in clock cycles. The effects of data and instruction cache memory latencies on timing analysis are not considered in this example. This is because we are not trying to build an accurate model of a microprocessor but instead illustrate the benefits of analysing branches rather than ignoring the effects of branch prediction. The simulator model features an in-order instruction pipeline with five pipeline stages that has a peak issue rate of four instructions per clock cycle.

<table>
<thead>
<tr>
<th>( b_i \rightarrow b_j )</th>
<th>( t_j )</th>
<th>( \delta_{ij} )</th>
<th>( x_{ij} )</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_1 )</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>( b_1 \rightarrow b_2 )</td>
<td>14</td>
<td>-9</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>( b_2 \rightarrow b_3 )</td>
<td>29</td>
<td>-8</td>
<td>18</td>
<td>378</td>
</tr>
<tr>
<td>( b_2 \rightarrow b_4 )</td>
<td>8</td>
<td>-7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( b_2 \rightarrow b_3 )</td>
<td>25</td>
<td>1</td>
<td>2</td>
<td>52</td>
</tr>
<tr>
<td>( b_2 \rightarrow b_4 )</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( b_3 \rightarrow b_5 )</td>
<td>11</td>
<td>-9</td>
<td>19</td>
<td>38</td>
</tr>
<tr>
<td>( b_3 \rightarrow b_5 )</td>
<td>7</td>
<td>-1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>( b_4 \rightarrow b_5 )</td>
<td>10</td>
<td>-5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( b_5 \rightarrow b_2 )</td>
<td>10</td>
<td>-5</td>
<td>17</td>
<td>85</td>
</tr>
<tr>
<td>( b_5 \rightarrow b_2 )</td>
<td>8</td>
<td>-1</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>( b_5 \rightarrow b_6 )</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>18</td>
</tr>
</tbody>
</table>

Total WCET 606

(a) Execution times in clock cycles  (b) Control flow graph

Figure 5.3: Timing analysis for a conditional loop construct

Listing C.1 on page 245 shows the C source code of the program used in this example. For the scope of this example we only consider the execution time of the code located within the function while_loop. The remaining code in the program represents our test harness. The function itself is based on the example we have discussed earlier in Section 4.4.2 on page 136, but uses additional statements in the then path of the conditional construct to fulfil the condition
stated in Theorem 4.2 on page 144. In addition, the number of loop iterations is a constant value greater than zero instead of being provided as a function parameter. Thus, the loop is now guaranteed to iterate at least once and the original branch instruction in basic block \( b_1 \) is no longer required to check the loop condition prior to executing the first iteration of the loop body.

The control flow graph \( G(V, E) \) corresponding to the example function is depicted in Figure 5.3(b) on the previous page. The whole program is translated by the GNU C Compiler version 2.6.3 to generate an executable image for the sim-outorder simulator. Figure C.1 shows the assembler code generated for the function \texttt{while_loop}. The program is executed using four different sets of sample data:

1. even/odd values alternating, with even value first;
2. even/odd values alternating, with an odd value first;
3. only even values; and
4. only odd values.

The table in Figure 5.3(a) on the preceding page provides an overview of all pairs of basic blocks \( b_i, b_j \in V \) that have a transition defined in the control flow graph, i.e. \( (b_i \rightarrow b_j) \in E \). Note that the transitions shown in the first column of the table distinguish between mispredicted and correctly predicted branches. Capturing only the effects of pipeline overlap across pairs of basic blocks is sufficient here because there are no long-running instructions present in the code. For example, some floating-point instructions are long-running instructions that take several clock cycles to execute and therefore may require analysis of the effects of pipeline overlap across multiple blocks. This would increase the complexity of the timing analysis.
The second column in the table shows the execution time in clock cycles for each basic block \( b_j \) in isolation, i.e. without taking into account any overlapping across adjacent blocks. The pipeline occupation of the predecessor \( b_i \) of basic block \( b_j \) has an impact on the amount of pipeline overlap between these two blocks. This amount of overlap, \( \delta_{ij} \), provided in the third column of the table is calculated using Equation (5.4) defined earlier.

The program flow analysis is very simple since the program flow only depends on the input data set being selected – no execution count constraints, such as loop bounds, have to be determined. The last column of the table in Figure 5.3(a) provides the execution count \( x_{ij} \) for each transition in the control flow graph, presuming that all values in the array are even. In this case, only the then path of the conditional statement is executed. According to the condition stated in Theorem 4.2, this program flow provides the WCET if the execution time difference between the then and else paths is at least twice the misprediction penalty. We will first show that this condition is met and then calculate the total WCET for the function.

The estimated WCET for the two execution paths including a misprediction caused by the branch instruction in basic block \( b_2 \) is given by:

\[
\tilde{T}(p_{\text{then}}) = t_2 + (t_3 + \delta_{23}) + (t_5 + \delta_{35}) \\
= 14 + (25 + 1) + (11 - 9) \\
= 42
\]

\[
\tilde{T}(p_{\text{else}}) = t_2 + (t_4 + \delta_{24}) + (t_5 + \delta_{45}) \\
= 14 + (5 + 1) + (10 - 5) \\
= 25
\]

In order to verify whether the condition is met we have to determine first how the misprediction penalty \( \delta \) decreases the pipeline overlap across the blocks involved
in the two paths. For this purpose, we also estimate the execution time for the two execution paths without branch misprediction:

\[
T(p_{\text{then}}) = t_2 + (t_3 + \delta_{23}) + (t_5 + \delta_{35}) \\
= 14 + (29 - 8) + (11 - 9) \\
= 37
\]

\[
T(p_{\text{else}}) = t_2 + (t_4 + \delta_{24}) + (t_5 + \delta_{45}) \\
= 14 + (8 - 7) + (10 - 5) \\
= 20
\]

We can observe that for both paths the execution time difference between the mispredicted and correctly predicted case is five cycles. This difference represents the actual penalty on the pipeline overlap caused by a branch misprediction. It is slightly higher, or in other words more pessimistic, than the original three cycle misprediction penalty that is used in the simulator configuration. Nevertheless, the condition in Theorem 4.2 is fulfilled since:

\[
|T(p_{\text{then}}) - T(p_{\text{else}})| \geq 2\delta_a \\
\iff 17 \geq 2 \cdot 5 \\
\iff 17 \geq 10
\]

Finally, we use Equation (5.6) to calculate the total WCET of the function \texttt{while_loop} from the values shown in Figure 5.3(a) on page 175. This gives us an estimated WCET of 606 clock cycles.

\[
\tilde{T}(G) = \sum_{\forall (b_i \rightarrow b_j) \in E} x_{ij} \cdot (t_j + \delta_{ij}) \quad (5.6)
\]

Table 5.1 on the facing page provides the measured execution time in clock cycle for each of the four data sets using different branch misprediction penalties, and
a bimodal and worst-case (wc) predictor configuration. The different measured execution times for the two data sets consisting of alternating values, i.e. the first and the second data set, are due to the fact that the behaviour of the bimodal branch predictor depends on the direction of the first branch in the sequence. For the second data set each branch is mispredicted, while only every second branch is mispredicted for the first data set.

<table>
<thead>
<tr>
<th>mpp</th>
<th>Data Set (bimod)</th>
<th>Data Set (wc)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>446</td>
<td>496</td>
</tr>
<tr>
<td>4</td>
<td>459</td>
<td>519</td>
</tr>
<tr>
<td>5</td>
<td>472</td>
<td>542</td>
</tr>
<tr>
<td>6</td>
<td>485</td>
<td>565</td>
</tr>
<tr>
<td>7</td>
<td>498</td>
<td>588</td>
</tr>
<tr>
<td>8</td>
<td>511</td>
<td>611</td>
</tr>
<tr>
<td>9</td>
<td>524</td>
<td>634</td>
</tr>
<tr>
<td>10</td>
<td>537</td>
<td>657</td>
</tr>
</tbody>
</table>

Table 5.1: Measured execution time in clock cycles for various misprediction penalties

Compared with the measured execution time of 576 cycles for the bimodal predictor configuration, the pessimism of the WCET estimate obtained by Equation (5.6) is 5.2%. Note that the relevant execution time figures are highlighted in Table 5.1. If we assume a misprediction for each executed branch instruction (wc – worst-case predictor) the measured execution time is 779 cycles, which represents an increase of 35.2% compared to the execution time measured for a bimodal branch predictor. Thus, the amount of pessimism is reduced by 30% by using our static analysis approach.

The execution time for the alternating behaviour of the conditional statement
exceeds the execution time of the *then* path if we use a misprediction penalty of at least eight cycles. For an eight cycle misprediction penalty, the actual penalty \( \delta_a \) is ten cycles. In this case, the measured execution times are 611 and 591 cycles, respectively, and the condition in Theorem 4.2 on page 144 is no longer met, since \( \delta_a > 8.5 \):

\[
2\delta_a \leq 17 \\
\iff \delta_a \leq 8.5
\]

However, this actual misprediction penalty would still satisfy the condition stated originally by Colin and Puaut (1999), but the simulation results confirm our earlier finding that their condition leads to unsafe WCET estimates if the following inequality is true:

\[
\delta_a \leq |T(p_{\text{then}}) - T(p_{\text{else}})| < 2\delta_a
\]

### 5.4 Program Flow Analysis

For the program flow analysis, an approach is used based on *Implicit Path Enumeration Technique* (IPET), which models the control flow of a program as a sequence of constraints on the execution count variables \( x_i \) (i.e. the number of times \( b_i \) is being executed) for each basic block in the control flow graph (Li and Malik, 1995; Puschner and Schedl, 1995). The WCET of a program represented by its control flow graph \( G \), in the following denoted by \( T(G) \), can then be calculated by finding the maximum value of the following equation:

\[
T(G) = \sum_{\forall i \cdot b_i \in V} x_i \cdot t_i
\]  

(5.7)

where \( t_i \) is the maximum number of clock cycles required to execute \( b_i \) and \( G = (V, E) \) is the control flow graph of the program under analysis.
A major limitation of Equation (5.7) is that it does not take into account the pipeline overlap between instructions located across the boundary of basic blocks. Therefore, using this formula results in a significant overestimation of the WCET for microprocessors using instruction pipelines.

In the following, we will address this problem by defining the execution time of a basic block depending on its preceding blocks and possible branch misprediction events that might occur due to the transition between the blocks. For this purpose, we will use the execution count of edges instead of basic blocks for the final calculation of the WCET and show how bounds on the maximum number of branch mispredictions can be integrated into the overall WCET analysis.

Each edge \((b_i \rightarrow b_j) \in E\) in the control flow graph is assigned with a variable \(d_{ij}\), which represents the number of times this edge is being followed. Furthermore, for each basic block \(b_i\) that contains a branch instruction we can distinguish between the execution counts for mispredicted (denoted by \(mp\)) and correctly predicted (denoted by \(cp\)) outgoing transitions:

\[
d_{ij} = d_{ij}^{mp} + d_{ij}^{cp},
\]

with \(\forall i, j \cdot (b_i \rightarrow b_j) \in E \land b_i \in V_c\)

5.4.1 Analysis of Loop Statements

As mentioned earlier, the pattern \(T^{n-1}N\) represents the behaviour of a branch instruction associated with a loop condition, assuming that the loop iterates \(n\) times and is repeated \(m\) times. According to Table 4.2 on page 127, an initial state of strongly not-taken (\(SN\)) is assumed as this equates to the worst-case initial state for a bimodal predictor. From this table, we can also derive the maximum number of branch mispredictions, denoted by \(mp_i\), that can be expected for the branch in \(b_i\) associated with the test of the loop condition. We can state the
following linear constraint on $mp_i$:

$$mp_i = d_{ij}^{mp} + d_{i,i+1}^{mp} \quad (5.9)$$

In general, the *not-taken* path of the branch instruction in $b_i$ is always mispredicted upon each loop exit, thus:

$$d_{i,j+1}^{mp} = m, \quad (5.10)$$

where $m$ is the number of times the loop construct itself is repeated.

### 5.4.2 Analysis of Conditional Statements

The behaviour of a branch instruction associated with a conditional statement cannot be determined statically if it depends solely on input data available only during run-time. In this case, we have to assume that all instances of the branch instruction are mispredicted, which according to Table 4.2 corresponds to an alternating behaviour of the branch. Thus, we can define the following linear constraints:

$$d_{ij}^{mp} = \left\lfloor \frac{n}{2} \right\rfloor$$

$$d_{i,i+1}^{mp} = n - d_{ij}^{mp}$$

We assume here that the WCET of the *not-taken* path exceeds that of the other path. Otherwise, $d_{ij}^{mp}$ and $d_{i,i+1}^{mp}$ have to be interchanged with each other.

However, according to the condition stated earlier in Theorem 4.2, we need to consider the path with the highest WCET in the calculation (instead of the alternating paths case), when the execution time difference $\lambda$ between the two paths of the conditional statement is at least twice the misprediction penalty $\delta$. Then, the corresponding pattern of the branch is either $N^n$, i.e. always *not-taken*, or $T^n$, i.e. always *taken*. According to Table 4.2 on page 127 and our assumption
that the execution time of the *not-taken* path exceeds that of the *taken* path, we can conclude that the maximum number of branch mispredictions is two for the edge associated with the *not-taken* path and zero for the other:

\[
\begin{align*}
d_{ij}^{mp} &= 0 \\
d_{i,i+1}^{mp} &= 2
\end{align*}
\]

The condition stated in Theorem 4.2 on page 144 needs to be evaluated for each conditional statement prior to establishing the set of ILP constraints for the WCET calculation.

### 5.4.3 WCET Calculation

We can restate the ILP cost function provided in Equation (5.7) such that the effects of instruction pipelining and branch prediction are taken into account:

\[
\tilde{T}(G) = \sum_{(i,j) \in E} (d_{ij}^{cp} \cdot (t_j + \delta_{ij}) + d_{ij}^{mp} \cdot (t_{mp}^{ij} + \delta_{ij}^{mp}))
\]  

(5.11)

This is a general calculation formula for estimating a WCET figure in the presence of an instruction pipeline and branch prediction, independent of how the individual analyses are actually performed. The scalability of the ILP problem to account for more complex control flow graphs is always of concern. For our approach, the increase in the number of ILP constraints is proportional to the number of branch blocks in the control flow graph. This relationship is achieved by pre-determining the worst-case number of branch mispredictions. In other ILP-based analysis approaches, e.g. Mitra et al. (2002), each branch instruction and each possible execution pattern have to be modelled as a separate ILP constraint which leads to a much higher number of constraints. Nevertheless, we acknowledge that they model a complex global-history predictor.
Figure 5.4: Applying a bimodal predictor to a conditional statement
5.5. CASE STUDY: ILP-BASED WCET ANALYSIS

Figure 5.4 on the facing page illustrates, for each of the four sample loop configurations we have discussed earlier in Chapter 4, the execution time behaviour per taken rate for an in-order instruction pipeline assuming a misprediction penalty of three and 17 clock cycles, respectively. In addition, the bottom two plots in this figure show the number of branch mispredictions per taken rate. The execution time figures have been calculated by using Equation (5.11) with the values provided in the table in Figure 5.3(a) on page 175.

For both the taken path biased (TP-A and TP-B) and the not-taken path biased (NP-A and NP-B) figures, the number of branch mispredictions is highest for a taken rate of 50% (alternating behaviour of the branch instruction). The worst-case misprediction rate reaches 100% in case of the not-taken path biased sample configuration and a 50% taken rate.

Complementary to Figure 5.4 on the facing page, Figure 5.5 on the next page shows the execution time behaviour for the NP-A and NP-B sample loop configurations depending on the misprediction penalty. The initial branch predictor state, weakly taken (WT) or weakly not-taken (WNT), and the branch pattern (alternating or always not-taken) is provided for each of the six plots.

For the NP-A sample, it can be observed from Figure 5.5 that the scenario giving rise to the estimated WCET figure changes from always not-taken to alternating when the branch misprediction penalty is seven or more clock cycles, as indicated by the left dashed vertical line. Similarly, the WCET scenario changes for the NP-B sample loop configuration when the branch misprediction penalty is 14 or more clock cycles.

5.5 Case Study: ILP-based WCET Analysis

Figure 5.6 on page 187 depicts the control flow graph G(V,E) for a conditional statement that is embedded within a loop. This example is based on the sample
Figure 5.5: Execution time behaviour depending on misprediction penalty function discussed earlier in Section 5.3. Again, Listing C.1 on page 245 shows the corresponding C source code for this example.

The set of branch blocks is defined by $V_c = \{b_2, b_3, b_5\}$. Basic blocks $b_2$ and $b_5$ contain conditional branches linked with the conditional statement and the loop condition, respectively, and $b_3$ contains an unconditional branch. The following set of linear constraints imposed by the program structure for the execution count of each basic block can be observed from the control flow graph:

\[
\begin{align*}
    x_1 &= d_{12} \\
    x_2 &= d_{12} + d_{52} = d_{24} + d_{23} \\
    x_3 &= d_{23} = d_{35} \\
    x_4 &= d_{24} = d_{45} \\
    x_5 &= d_{35} + d_{45} = d_{52} + d_{56} \\
    x_6 &= d_{56}
\end{align*}
\]
Table 5.2 on the following page provides the execution counts $d_{ij}^m$ and $d_{ij}^{mp}$ derived using the analysis results in Table 4.2 on page 127, the execution time $t_j$ of the basic block $b_j$, and the amount of overlap $\delta_{ij}$ between $b_i$ and $b_j$ for each transition $(b_i \rightarrow b_j) \in E$ in the control flow graph. Although the execution time values in this table were obtained from the SimpleScalar architectural simulator (Austin et al., 2002), any other source of basic block execution time estimates could be used as well. For this example, we have configured the simulator to use an in-order instruction pipeline without caches.

![Figure 5.6: Annotated control flow graph](image)

Providing a loop bound is essential in order to make the ILP problem decidable. The loop in the control flow graph can be identified by the fact that basic block $b_2$ dominates $b_5$ (i.e. all paths from the start vertex to basic block $b_5$ pass through $b_2$) and there is also a back edge $(b_5 \rightarrow b_2) \in E$ between these two blocks. Therefore, basic block $b_2$ represents the header of the loop, which we assume iterates $n = 20$ times. Furthermore, we assume that the function itself is
executed only once, thus:

\[ x_1 = 1 \quad \land \quad x_2 = 20 \cdot x_1 \quad \land \quad x_5 = x_2 \]

The branch instruction in \( b_5 \), which is associated with the loop condition, has the pattern \( T^{n-1}N \) and we assume in the following that \( n > 3 \). According to Equations (5.9) and (5.10), we can define two additional linear constraints:

\[ d^{mp}_{52} = 2 \quad \land \quad d^{mp}_{56} = 1 \]

We assume that the behaviour of the conditional statement is not known at compile-time and, therefore, we are not able to define exact values for the execution count variables \( x_3 \) and \( x_4 \) representing the two paths of the conditional statement. However, we can make reasonable worst-case assumptions about the behaviour of the branch instruction by taking into account the condition stated previously in Definition 4.2 on page 144.

If this condition is met we need to consider the path with the highest WCET in the calculation (instead of the alternating paths case), which in our example is the \textit{then} path represented by basic block \( b_3 \). In order to verify the condition
we have to determine how much the misprediction penalty decreases the pipeline overlap between the blocks involved in the two paths. For this purpose, we also estimate the WCET difference between the mispredicted, \( \tilde{T}(p) \), and non-mispredicted case, \( T(p) \), for each of the two possible execution paths, which for our example is five clock cycles for each path:

\[
\tilde{T}(p_{\text{then}}) - T(p_{\text{then}}) = (t_3^{mp} + \delta_{23}^{mp}) - (t_3 + \delta_{23}) \\
= (25 + 1) - (29 - 8) = 5 \\
\tilde{T}(p_{\text{else}}) - T(p_{\text{else}}) = (t_4^{mp} + \delta_{24}^{mp}) - (t_4 + \delta_{24}) \\
= (5 + 1) - (8 - 7) = 5
\]

This execution time difference represents the actual misprediction penalty, \( \delta_a \), on the pipeline overlap caused by a branch misprediction in basic block \( b_2 \). It should be noted that for our example the actual penalties for the then and else paths are the same but this is not necessarily the case in general. The actual penalty is slightly higher than the three cycle misprediction penalty purely associated with the branch misprediction. This is due to the fact the branch prediction also causes the pipeline to stall. The analysis shows that the condition stated in Definition 4.2 on page 144 is fulfilled:

\[
|T(p_{\text{then}}) - T(p_{\text{else}})| \geq 2\delta_a \\
\iff 17 \geq 2 \cdot 5 \\
\iff 17 \geq 10
\]

In this case, the corresponding execution pattern of the branch instruction in basic block \( b_2 \) is \( N^n \), i.e. always not-taken, and according to Table 4.2, the maximum number of mispredictions for this branch is two. Furthermore, we assume that the target address of the unconditional branch instruction in basic block \( b_3 \) is mispredicted on its first execution. We can define the following additional ILP
constraints for this case:

\[ x_4 = 0 \land d_{35}^{mp} = 1 \land d_{23}^{mp} = 2 \]

**Considering branch interference.** Based on the classification defined in Table 4.2 the branch instruction in \( b_5 \) is *taken* biased while the branch in \( b_2 \) is *not-taken* biased. Whilst the case study presented here is probably too small to exhibit extensive branch interference in practice, let us assume that the branches in basic blocks \( b_2 \) and \( b_5 \) are mapped to the same BHT entry and thus their predictor behaviour interferes with each other. In this case, the actual branch behaviour seen by the predictor alternates between the *taken* (due to basic block \( b_5 \)) and the *not-taken* (due to basic block \( b_2 \)) directions, which, in the worst-case, causes both branches to be always mispredicted. As there is a significant impact on performance due to this interference we certainly want to remove it rather than model it.

**WCET calculation.** Using Equation (5.11) we can now calculate the total WCET of our example function from the values provided in Table 5.2 by using an ILP problem solving program (e.g. `lp_solve`). This gives us an estimated WCET of 606 clock cycles assuming a branch misprediction penalty of three clock cycles and \( n = 20 \) loop iterations. The pessimism of this WCET estimate compared with the measured execution time of 576 cycles is 5.2%.

The main reason for this overestimation is the use of maximum values for \( t_j \), \( \delta_{ij} \), \( t_j^{mp} \), and \( \delta_{ij}^{mp} \) for any transition \( (b_i \rightarrow b_j) \in E \). Taking into account that these values may actually vary, depending on *some* basic block preceding \( b_i \), would reduce the overestimation but increase the complexity of the WCET analysis significantly. The reason is it would require to model instruction pipeline effects along block sequences of arbitrary length. The amount of pessimism in this case

190
is independent of the number of loop iterations due to a constant overestimation of the loop body.

If we assumed that the branch instruction in basic block $b_2$ alternates between its *taken* and *not-taken* directions and is always mispredicted (see Table 4.2) the estimated WCET would have been 526 clock cycles. In comparison with 576 clock cycles (see above), this figure does clearly not represent the actual WCET. Hence, simply assuming the worst-case number of branch mispredictions and an alternating behaviour of the branch instruction provides an unsafe WCET estimate in this case.

Alternatively, a conservative and simplistic approach would be to take into account both the maximum number of branch mispredictions and the longest execution path of the conditional statement. It should be noted that the ILP problem solving program automatically assumes this scenario if no constraints are provided on $d_{23}^{mp}$ and $d_{24}^{mp}$ (and hence on $x_3$ and $x_4$) and, as a result, increases the pessimism of the WCET estimate to 20.8%. In fact, this scenario is not even possible for a bimodal branch predictor because a branch instruction cannot always have the same outcome and be mispredicted at the same time.

5.6 Summary

The integration of individual WCET analysis methods is not straightforward due to the interaction between the analyses. This often leads to large increases in computational complexity, the introduction of unnecessary pessimism, or even to unsafe WCET estimates.

We have shown in the course of this chapter how a previously published approach for static WCET analysis of dynamic branch predictors (Bate and Reuteumann, 2004), which also has been discussed in more detail in Chapter 4, can be integrated with instruction pipeline analysis and the WCET be estimated using
an ILP-based calculation method. This is achieved by first calculating the number of branch mispredictions and then representing these as constraints. Taking this approach results in significantly fewer constraints than for other approaches estimating misprediction numbers as part of the ILP problem. The reason is that the constraints do no longer need to be formulated for the various predictor state transitions. Hence the analysis approach presented in this chapter is more scalable.
Chapter 6

Global-History Branch Predictors

This chapter continues with the static analysis of dynamic branch predictors, but the scope of the analysis is now extended to more complex predictor configurations. Such predictors include two-level adaptive predictors that use global branch history for predicting the direction of branches. Global branch history records the outcome of all branch instructions executed recently by the microprocessor in a single branch pattern. In contrast, local branch history uses the branch address to associate the branch pattern with a particular branch instruction or set of branch instructions.

In this chapter, Section 6.1 defines the configuration of the global branch predictor that is used as reference throughout this chapter and defines the basic terminology. A simple example of a for-loop construct with a single branch instruction is discussed in Section 6.2. The example is used to identify three different types of history patterns. Based on the characteristics of these pattern types, an analysis approach is presented in order to estimate an upper bound on the number of branch mispredictions that can be expected for such control structures. This approach has been published earlier in the second part of a ECRTS 2004 research paper by Bate and Reutemann (2004). The section concludes with the discussion of a previously published code example and provides a more de-
tailed explanation of the obtained results. Section 6.3 presents a more complex example where the history pattern space has to be analysed for a loop construct containing a single conditional statement. Finally, Section 6.4 summarises the findings of this chapter.

6.1 Branch Patterns

In the course of this chapter, we assume a two-level adaptive predictor using global branch history. Figure 6.1 depicts the configuration of this predictor.

![Figure 6.1: Global history two-level adaptive predictor (GAg)](image)

The branch history register (BHR) of a GAg two-level adaptive predictor (see Figure 6.1) stores a pattern representing the outcome of recent branch instructions. The structure of this pattern is formally defined in the following:

**Definition 6.1 (Branch History Pattern)**

The branch history pattern of a global branch predictor is the finite sequence \( \langle h_k, \ldots, h_1 \rangle \), with \( h_i \in \{0, 1\} \), of not-taken (0) and taken (1) instances that captures the resolved outcome of the most recent \( k \) branch instructions executed by the processor.

We use 0 and 1 instead of \( N \) and \( T \) to represent the not-taken and taken branch direction, respectively, in order to indicate that this pattern is used in a real
data structure of the microprocessor. The rightmost bit $h_1$ of the branch history register represents the outcome of the most recent branch instance. The branch history register has a size of $k$ bits, these bits are used to select one of the $2^k$ two-bit counters stored in the pattern history table (PHT). For a GAg predictor, both the branch history register and the pattern history table are the same for all static branch instructions. Thus, there is no direct mapping between the address of branch instructions and the predictor state, unlike the case for the bimodal predictor analysed in the previous chapter. This complicates the static analysis of branch predictor performance because the set of possible branch patterns has to be derived from the combined static branch execution patterns of all recent branches. A static branch execution pattern exists if we can determine the behaviour of a branch from its semantic context, i.e. the branch is easy to predict. Based on the static branch execution pattern of a for-loop we can define the set of possible branch patterns for this construct as follows:

**Definition 6.2 (Loop Branch Pattern Space)**

The loop branch pattern space of a static branch execution pattern $\langle d_1, \ldots, d_n \rangle$, with $d_1, \ldots, d_{n-1} = T$ and $d_n = N$, is defined as the set of patterns

\[
\langle d_1, d_2, \ldots, d_{n-1}, d_n \rangle \\
\langle d_2, d_3, \ldots, d_n, d_1 \rangle \\
\vdots \\
\langle d_n, d_1, \ldots, d_{n-2}, d_{n-1} \rangle.
\]

(6.1)

The static branch execution pattern $\langle T^{n-1}, N \rangle$ used in this definition characterises the behaviour of a for-loop construct. All patterns in the branch pattern space are unique and their number is equal to the length $n$ of the static branch execution pattern, i.e. the number of loop iterations.
CHAPTER 6. GLOBAL-HISTORY BRANCH PREDICTORS

Listing 6.1: C source code for a for-loop construct containing an if-statement

```c
int_t i;

while(1) {
    for (i = 0; i < N; i++) {
        /* do something */
    } /* end for */
} /* end while */
```

6.2 Analysis of Simple Branch Patterns

Let us consider a simple nested for-loop construct in order to illustrate how the space of possible global history patterns can be explored. Figure 6.2 shows the control flow graph of the nested loop construct provided in Listing 6.1.

Figure 6.2: Control flow graph for a nested loop construct

The inner loop of this construct is iterated a fixed number of times, \( n \). We assume that the inner loop only contains sequential statements, i.e. there are no further branch instructions to be taken into consideration. Then, an outer loop continually repeats the inner loop. We further assume that the outer loop iterates using an unconditional branch instruction such that no additional conditional branch has to be recorded in the global BHR. The purpose of this is to restrict the following analysis of the branch predictor behaviour to only one branch. Alternatively, we could use a PAp local-history predictor instead of a global-history predictor in order to overcome this assumption but with the same effect.
6.2. ANALYSIS OF SIMPLE BRANCH PATTERNS

on the overall branch predictor state.

6.2.1 Deriving Branch History Patterns

Table 6.1 on the following page shows the resulting branch history patterns of length $k = 4$ for $n = 3$, $n = 4$, and $n = 6$ loop iterations for our nested loop example. The second sub-column of each column in the table shows the current predictor history pattern prior to the execution of the branch instruction. The third sub-column represents the resolved outcome of the branch and the history pattern in the following line is updated accordingly. Like in the previous chapter, our aim is now to find conditions where we have biased predictor states, i.e. the outcome of branches is easy to predict at compile-time. For this type of loop construct we are primarily interested in the predictor being biased towards the \textit{taken} direction because this is the most frequent direction of loop control branches.

If we assume that the global history pattern is initially set to $\langle 0000 \rangle$, then the first $k - 1$, in this case three, iterations of the loop produce history patterns that occur only once for any number of repetitions of the loop. This number stems from the fact that for a history of length $k$ there are $k - 1$ different patterns with more than one leading consecutive zeros. These first $k - 1$ branches are all mispredicted because in the worst-case the two-bit counter of each pattern is initially in the \textit{strongly not-taken} state and therefore at least two consecutive \textit{taken} branches are required to predict the outcome of the \textit{taken} branches correctly. For the single \textit{not-taken} branch, however, we assume that the predictor is in the \textit{strongly taken} state. In the context of this thesis we refer to this type of history patterns as \textit{non-repeating history patterns}.

After the non-repeating history patterns, only patterns with at most one consecutive zero occur in the global branch history register and the patterns
### Table 6.1: History patterns for a simple loop construct

<table>
<thead>
<tr>
<th>$n = 3$</th>
<th>$n = 4$</th>
<th>$n = 6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Pattern</td>
<td>dir</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>0001</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>0011</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0110</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0110</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0110</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
repeat themselves after a number of iterations. In order to determine how many different history patterns exist we have to distinguish between two cases.

1. \( (n \leq k) \): In this case, the branch history register records all possible loop branch patterns of length \( n \). Therefore, there exist \( n \) different branch patterns, which we call *repeating history patterns*. There is a one-to-one correspondence between the history pattern for a particular iteration and the branch outcome. Therefore, branches are predicted correctly after the corresponding patterns occurred twice and the maximum number of branch mispredictions is \( 2n \).

It should also be noted that in the first case the global history predictor is also able to predict the *not-taken* branch instances correctly because a unique branch pattern exists for these branches.

2. \( (n > k) \): The number of consecutive *taken* branches exceeds the length of the history pattern and the *not-taken* branch disappears from the history pattern. There are now two types of history patterns: The *repeating history patterns* always contain a single zero representing a *not-taken* branch. There are \( k \) ways to place the single zero in the branch history pattern of length \( k \). Thus, the number of repeating history patterns for this case is \( k \) and the maximum number of mispredictions is \( 2k \). After these patterns have occurred, the branch history loses track of the *not-taken* instance. This results in a pattern with all ones and this pattern may now occur for different iterations of the loop. In particular, the pattern occurs for both *taken* and *not-taken* branches. We refer to this pattern as *biased history pattern* because of the history being biased towards the *taken* direction. This pattern occurs for the last \( n - k \) branches during each repetition of the loop construct. These branches share the same entry in the pattern history table and, therefore, the behaviour of the predictor is equal to a bimodal predictor.
### 6.2.2 Bounding the Number of Mispredictions

Table 6.2 summarises our findings about the three different types of history patterns that can occur during the execution of the `for`-loop discussed in this section. These results are applicable to nested loop constructs without any additional control structure embedded.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Frequency</th>
<th>Example</th>
<th>Mispredictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Repeating</td>
<td>$k - 1$</td>
<td>⟨0001⟩</td>
<td>$k - 1$</td>
</tr>
</tbody>
</table>
| Repeating   | min($n, k$) | ⟨1101⟩  | \[
\begin{align*}
2n, & \text{ if } n - k \leq 0 \\
2k, & \text{ if } n - k > 0 
\end{align*}
\]
| Biased      | \[
\begin{align*}
0, & \text{ if } n - k \leq 0 \\
n - k, & \text{ if } n - k > 0 
\end{align*}
\] | ⟨1111⟩ | \[
\begin{align*}
0, & \text{ if } n - k \leq 0 \\
2, & \text{ if } n - k = 1 \\
2m, & \text{ if } n - k = 2 \\
3 + m, & \text{ if } n - k = 3 \\
2 + m, & \text{ if } n - k \geq 4 
\end{align*}
\]

Table 6.2: Global history pattern types for the `for`-loop example

In order to derive an upper bound on the number of mispredictions for the loop construct in our example we have to distinguish between five cases, which are broken down in the last column of Table 6.2. The upper bound on the number of mispredictions for the biased history pattern can be obtained by replacing $n$ by the expression $n - k$ in Theorem 4.1 because this pattern uses a single entry in the PHT. We can then combine the misprediction bounds for each pattern type provided in Table 6.2 to state the following theorem:

**Theorem 6.1 (Global history loop)**

Let $n$ be the number of loop iterations, $k$ be the length of the global branch history table, and $m$ be the number of times the loop is repeated. Then, the upper bound on the number of branch mispredictions for the repeated execution
6.2. ANALYSIS OF SIMPLE BRANCH PATTERNS

of a simple loop construct is given by:

\[
mp_{\text{loop}}(n, k, m) = \begin{cases} 
2n + k - 1, & \text{if } n - k \leq 0 \\
3k + 1, & \text{if } n - k = 1 \\
3k + 2m - 1, & \text{if } n - k = 2 \\
3k + m + 2, & \text{if } n - k = 3 \\
3k + m + 1, & \text{if } n - k \geq 4
\end{cases}
\]

Although the loop construct in this example involved only a single branch instruction the static analysis of the global history branch predictor behaviour is more complex than that of the bimodal predictor used in the previous chapter. Also, we simplified the exploration of the pattern space by assuming that the branch associated with the outer loop is unconditional and therefore does not need to be considered in the global history.

It should be noted from Theorem 6.1 that there is a significant increase in the upper bound on the number of mispredictions from \((n - k) = 1\) to \((n - k) = 2\). Furthermore, for \((n - k) \geq 2\) the misprediction bound is proportional to the number of iterations of the outer loop and is completely independent from the number of times the inner loop iterates.

This supports the experimental results for the Intel Pentium III (Intel, 1999), AMD Athlon (AMD, 2002), and UltraSPARC III (Horel and Lauterbach, 1999) microprocessors reported in Engblom (2002). He evaluates the execution times of a nested loop construct for different numbers of iterations of the inner loop. The number of iterations of the outer loop remains fixed. The results for the above mentioned microprocessors indicate that the average execution time per loop iteration decreases when the number of executed iterations increases. For the AMD Athlon microprocessor, the total execution time rises significantly between the ninth and tenth iteration of the inner loop. Similar results are reported for
the Pentium III and the UltraSPARC III microprocessors where the rise appears after the fourth and 13th iteration, respectively.

Unfortunately, Engblom does not provide an explanation as to why these effects occur. Using the branch predictor configuration of the AMD Athlon microprocessor as an example, we will provide a brief explanation of these effects in the following, combined with the findings of branch prediction behaviour for nested loop constructs discussed earlier.

The AMD Athlon uses a global history two-level predictor that is combined with a 2048-entry BTB (AMD, 2002). The global predictor uses eight bits of branch history and four bits of the branch instruction address to index the PHT. The loop control branch of the outer loop causes one additional branch to be recorded in the branch history. Another branch is recorded because of the initial check of the inner loop condition. This branch is usually not-taken and therefore the content of the BHR is the same as if these two additional branches were not present. According to Theorem 6.1, the maximum number of mispredictions can be expected for $n - k = 2$, which would suggest a history length of $k = n - 2 = 10 - 2 = 8$, which in fact represents the predictor configuration of the AMD Athlon microprocessor. Similar considerations apply to the Pentium III and UltraSPARC III microprocessors.

6.2.3 Experimental Evaluation

Figure 6.3 on page 204 shows the simulation results obtained from the SimpleScalar sim-outorder simulator (Austin et al., 2002) for the nested loop construct in Figure 6.2 that has already been discussed earlier in this chapter. The original simulator has been extended to generate an execution trace of all branch instructions in order to be able to analyse branch predictor behaviour on a per-instruction basis. The execution time results have been extracted from the
6.2. ANALYSIS OF SIMPLE BRANCH PATTERNS

pipeline trace output. We have configured the simulator to ignore the effects of data and instruction cache access latencies so that the effects of branch prediction can be isolated. The simulation has been executed for three different dynamic branch predictor configurations:

- **bimodal** – a 2-bit counter branch predictor using a 4096-entry BHT;

- **GAg** – a two-level adaptive predictor using a single 12-bit BHR to index a global 4096-entry PHT; and

- **Athlon** – a branch predictor based on the AMD Athlon microprocessor, which uses a global history two-level predictor that is combined with a 2048-entry BTB. The global-history branch predictor of the Athlon uses eight bits of branch history and four bits of the branch instruction address to index the PHT.

According to our analysis results presented in Chapter 4, the maximum number of branch mispredictions for the bimodal predictor occurs for an inner loop with two iterations. For more than two loop iterations the number of mispredictions remains constant. This is also indicated by the simulation results provided in Figure 6.3(b) on the following page, which shows the number of mispredictions for the three predictor configurations. Accordingly, Figure 6.3(a) on the next page shows a significant increase in the number of clock cycles for two loop iterations and then a steady increase of the execution time for more than two iterations. The simulator has not been configured to resemble an existing microprocessor architecture, but the results presented here for the AMD Athlon microprocessor have been confirmed by the author on a real Athlon microprocessor using its performance counters.
Figure 6.3: Simulation results
6.3 Analysing Loops with Conditional Statements

Listing 6.2 shows the C source code for a for-loop construct that contains a single if-statement. This example involves two branch instructions: the loop control branch and the branch instruction associated with the conditional statement. In contrast to the previous chapter, we can no longer analyse the behaviour of the two branch instructions independently from each other because the predictor combines their histories in a single (global) branch history register. In this example we only analyse the predictor behaviour for the loop control branch and assume that the outcome of the conditional statement cannot be determined at compile-time. This assumption can be relaxed if we can show that the conditional statement satisfies the condition stated in Theorem 4.2 on page 144 and, therefore, only one execution path has to be considered in the analysis.

Table 6.3 on the following page shows the branch history patterns for three different combinations of pattern length \( k \) and number of loop iterations \( n \). The bits denoted as \( x \) in a branch history pattern represent the outcome of the branch instances associated with the if-statement within the loop construct. The branch instruction executed during the current iteration of the loop is represented by the rightmost \( x \), whereas the outcome of previous branch instances are captured by the \( x \) instances to the left. Each \( x \) instance can either be taken or not-taken, and increases the number of possible patterns by a factor two. The variation of a branch history pattern from one loop iteration to the next is limited by the
### Table 6.3: History patterns for a loop containing a conditional statement

<table>
<thead>
<tr>
<th></th>
<th>Pattern</th>
<th>dir</th>
<th>Pattern</th>
<th>dir</th>
<th>Pattern</th>
<th>dir</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 4, k = 5</td>
<td>0000x</td>
<td>T</td>
<td>0000x</td>
<td>T</td>
<td>00000x</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>00x1x</td>
<td>T</td>
<td>00x1x</td>
<td>T</td>
<td>0001x</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>x1x1x</td>
<td>T</td>
<td>x1x1x</td>
<td>T</td>
<td>01x1x</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>x1x1x</td>
<td>N</td>
<td>x1x1x</td>
<td>T</td>
<td>11x1x</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>x1x0x</td>
<td>T</td>
<td>x1x1x</td>
<td>T</td>
<td>11x1x</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>x0x1x</td>
<td>T</td>
<td>x1x1x</td>
<td>N</td>
<td>11x1x</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>x1x1x</td>
<td>N</td>
<td>x0x1x</td>
<td>T</td>
<td>10x1x</td>
<td>T</td>
</tr>
<tr>
<td>n = 6, k = 5</td>
<td>x0x1x</td>
<td>T</td>
<td>x1x1x</td>
<td>T</td>
<td>01x1x</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>x1x1x</td>
<td>T</td>
<td>x1x0x</td>
<td>T</td>
<td>10x1x</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>x1x1x</td>
<td>N</td>
<td>x0x1x</td>
<td>T</td>
<td>10x1x</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>x1x1x</td>
<td>T</td>
<td>x1x1x</td>
<td>T</td>
<td>11x1x</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>x1x1x</td>
<td>N</td>
<td>x1x1x</td>
<td>N</td>
<td>11x1x</td>
<td>N</td>
</tr>
<tr>
<td>n = 6, k = 6</td>
<td>x1x1x</td>
<td>T</td>
<td>x1x1x</td>
<td>T</td>
<td>11x1x</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>x1x1x</td>
<td>T</td>
<td>x1x1x</td>
<td>T</td>
<td>11x1x</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>x1x1x</td>
<td>N</td>
<td>x1x1x</td>
<td>N</td>
<td>11x1x</td>
<td>N</td>
</tr>
</tbody>
</table>
6.3. ANALYSING LOOPS WITH CONDITIONAL STATEMENTS

fact that the pattern is shifted to the left for each iteration and, thus, only the bits being shifted into the pattern can change. However, this behaviour does not reduce the overall number of possible history patterns.

Like in our previous example, we can distinguish in Table 6.3 three different types of branch history patterns, which are: non-repeating, repeating, and biased history patterns. However, instead of only recording the loop branch in the branch history pattern, the predictor has to record an additional branch, which is associated with the conditional statement. This reduces the number of loop iterations that can be captured by the branch history pattern and, therefore, we have to change the findings in Table 6.2 on page 200 accordingly.

**Theorem 6.2 (Number of loop control branches)**

_in the case of two branch instances per loop iteration, a branch history pattern of length \(k\) can store at most \(\left\lfloor \frac{k}{2} \right\rfloor\) instances of the loop control branch._

**Proof:** For \(k = 2\), the pattern consists of \(\langle 1x \rangle\) or \(\langle 0x \rangle\), i.e. there is one loop control branch instance contained in the pattern. Now we assume that in general for some integer \(k \geq 2\) the number of loop control branch instances in a branch history pattern of length \(k\) is given by:

\[
lcb(k) = \left\lfloor \frac{k}{2} \right\rfloor
\]

If \(k\) is even, the branch history pattern can record one additional instance of the branch associated with the conditional construct but no additional loop control branch, thus

\[
lcb(k + 1) = \left\lfloor \frac{k + 1}{2} \right\rfloor = \left\lfloor \frac{k}{2} \right\rfloor = lcb(k) \quad \& \quad k \text{ even} \quad (6.2)
\]

If \(k\) is odd, one additional loop control branch can be recorded in the branch history pattern, therefore

\[
lcb(k + 1) = \left\lfloor \frac{k + 1}{2} \right\rfloor = \left\lfloor \frac{k}{2} \right\rfloor + 1 = lcb(k) + 1 \quad \& \quad k \text{ odd} \quad (6.3)
\]

207
Thus, there are \(\left\lfloor \frac{k}{2} \right\rfloor - 1\) different non-repeating history patterns. The outcome of the branch instances associated with the conditional construct is not relevant for this type of branch pattern because the patterns occur only once for a particular loop construct. Therefore, the maximum number of mispredictions for the non-repeating history pattern is given by:

\[
mp_{nrp}(n, k) = \left\lfloor \frac{k}{2} \right\rfloor - 1 \tag{6.4}
\]

The number of repeating history patterns depends on whether the branch history pattern can record the branch instances of all loop iterations or not. In the first case, i.e. \(n \leq \left\lfloor \frac{k}{2} \right\rfloor\), there are \(n\) different patterns generated by the outcome of the loop control branch (ignoring the outcome of the conditional statement). In order to determine the total number of possible patterns we have to distinguish between the pattern bits associated with the loop control branch and the bits associated with the outcome of the conditional statement. As mentioned earlier in this chapter, the outcome of the conditional statement increases the number of permutations of the branch pattern. A branch pattern of length \(k\) can record \(\left\lceil \frac{k}{2} \right\rceil\) instances of the conditional statement and each instance increases the number of patterns by a factor two. Each pattern is mispredicted twice and, therefore, the maximum number of mispredictions caused by the repeating branch history patterns is given by:

\[
mp_{rep}(n, k) = 2 \cdot n \cdot 2^{\left\lceil \frac{k}{2} \right\rceil} = n \cdot 2^{\left\lceil \frac{k}{2} \right\rceil + 1}, \quad \text{for } n \leq \left\lfloor \frac{k}{2} \right\rfloor \tag{6.5}
\]

In the second case, i.e. \(n > \left\lfloor \frac{k}{2} \right\rfloor\), the branch history pattern can only record the outcome of the two branch instructions involved in the loop for \(\left\lfloor \frac{k}{2} \right\rfloor\) loop iterations. Similar to the considerations for the first case, the maximum number
of mispredictions is:

\[
mp_{\text{rep}}(n, k) = 2 \cdot \left\lfloor \frac{k}{2} \right\rfloor \cdot 2^\left\lceil \frac{k}{2} \right\rceil
\]

\[
= \left\lfloor \frac{k}{2} \right\rfloor \cdot 2^\left\lceil \frac{k}{2} \right\rceil + 1, \quad \text{for } n > \left\lfloor \frac{k}{2} \right\rfloor
\] (6.6)

The remaining \( n - \left\lfloor \frac{k}{2} \right\rfloor \) iterations of the loop produce \textit{taken} biased history patterns, i.e. all bits associated with the loop control branch are set to 1. The number of possible patterns \( p_{\text{bip}} \) for this type of branch history pattern is given by Equation (6.7):

\[
p_{\text{bip}}(n, k) = \begin{cases} 
2^\left\lceil \frac{k}{2} \right\rceil, & \text{if } n > \left\lfloor \frac{k}{2} \right\rfloor \\
0, & \text{if } n \leq \left\lfloor \frac{k}{2} \right\rfloor
\end{cases}
\] (6.7)

In contrast to the previous example the instances of the loop control branch associated with this pattern type no longer share the same entry in the pattern history table. This is because of the \( 2^\left\lceil \frac{k}{2} \right\rceil \) permutations of the bits in the branch history pattern recording the outcome of the conditional statement. Depending on these bits different entries in the pattern history table are addressed.

<table>
<thead>
<tr>
<th>Ref</th>
<th>Pattern</th>
<th>Pattern (L C)</th>
<th>dir</th>
<th>State Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>11101010</td>
<td>1111 1000</td>
<td>T</td>
<td>( s_i^a \xrightarrow{T} s_{i+1}^a )</td>
</tr>
<tr>
<td>b</td>
<td>10101010</td>
<td>1111 0000</td>
<td>T</td>
<td>( s_i^b \xrightarrow{T} s_{i+1}^b )</td>
</tr>
<tr>
<td>b</td>
<td>10101010</td>
<td>1111 0000</td>
<td>N</td>
<td>( s_i^b \xrightarrow{N} s_{i+1}^b )</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>c</td>
<td>10111010</td>
<td>1111 0100</td>
<td>T</td>
<td>( s_i^d \xrightarrow{T} s_{i+1}^d )</td>
</tr>
<tr>
<td>a</td>
<td>11101010</td>
<td>1111 1000</td>
<td>N</td>
<td>( s_i^a \xrightarrow{N} s_{i+1}^a )</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>a</td>
<td>11101010</td>
<td>1111 1000</td>
<td>T</td>
<td>( s_i^a \xrightarrow{T} s_{i+1}^a )</td>
</tr>
</tbody>
</table>

Table 6.4: Example of a worst-case scenario for a biased history pattern

Table 6.4 illustrates a possible worst-case scenario for this type of branch history pattern. The second column in this table shows some patterns that may
occur during the execution of a loop construct with an embedded conditional statement. The third column splits the pattern into two sub-patterns: one representing the history of the loop control branch (L) and the other representing the history of the branch associated with the conditional statement (C). There are two different patterns (indicated as \(a\) and \(b\)) where the direction of the loop control branch changes for each instance of the pattern. We can continue the sequence of patterns such that we obtain an alternating sequence of \(t\)aken and \(n\)ot-t\(a\)ken instances of the loop control branch associated always with the same branch history pattern, i.e. \(s^a_i \rightarrow T \Rightarrow s^a_{i+1} \rightarrow N \Rightarrow s^a_{i+2} \rightarrow T \Rightarrow s^a_{i+3} \rightarrow N \Rightarrow \cdots\), where \(s^a_i \rightarrow T \Rightarrow s^a_{i+1}\) represents the transition of the branch predictor state due to the \(t\)aken outcome of the branch instruction \(a\).

As a consequence of this example, we have to assume that each biased branch history pattern causes a branch misprediction each time the loop control branch is executed.

### 6.4 Summary

This chapter has extended the static analysis approach presented in Chapter 4 for bimodal branch predictors to two-level adaptive predictors using global branch history as implemented, for example, in the AMD Athlon microprocessor. The number of mispredictions for simple control statements has been derived by exploring the global branch history pattern space. We have shown in the course of this chapter that determining an upper bound on the number of mispredictions for global-history branch predictors is complex even for very simple control statements.

Based on our work published previously in Bate and Reutemann (2004), we have explained the effects observed by Engblom (2002, 2003), which he called \(i\)nversions, for the Intel Pentium III, AMD Athlon, and UltraSPARC III micro-
6.4. SUMMARY

processors. For the AMD Athlon microprocessor, the total execution time rises significantly between the ninth and tenth iteration of the inner loop. In order to explain this behaviour, we have developed a theoretical model based on branch patterns for global history branch predictors. Assuming \( n \) iterations of a inner loop and a global branch history register size of \( k \) bits, the maximum number of branch mispredictions can be expected for \( (n - k) = 2 \), i.e. the number of iterations exceeds the branch history register size by two.

The experimental results supporting our theoretical model have been obtained by SimpleScalar using a branch predictor configuration that resembles the one of the AMD Athlon microprocessor. In addition, the results have been confirmed by the author on a real Athlon microprocessor using its performance counters.
Chapter 7

Coding and Compilation Techniques Supporting Static Analysis

This chapter presents coding and compilation techniques for reducing the pessimism associated with static execution time analysis of hard to predict branches. These branches have to be excluded from static analysis because the branch behaviour cannot be determined from their semantic context. In terms of static WCET analysis we have to assume that all hard to predict branch instructions are mispredicted.

For example, code optimisation guidelines exist for the AMD Athlon (AMD, 2002) and the Intel Pentium 4 (Intel, 2003) microprocessors. These guidelines propose various coding recommendations as to how the performance gained by hardware features such as caches and branch prediction can be improved. Although many of the proposed recommendations aim at improving the processing performance in the average case, they should be taken into account for static WCET analysis as far as possible. A general recommendation is that branches
with conditions depending on random input data should be avoided because of the significant number of mispredictions caused by such branches.

In this chapter, Section 7.1 provides a short introduction to a technique called predicated execution – a technique that can effectively reduce the performance penalties associated with control transfer instructions by eliminating them from the program code. Section 7.2 introduces the concept of loop unrolling, which also aims at the elimination or reduction of loop control branches. Section 7.3 presents a small case study that demonstrates how predication and loop unrolling can be applied in order to reduce the number of hard to predict branch instruction for one of the sample programs presented in Chapter 3. Finally, Section 7.4 concludes the chapter.

### 7.1 Predicated Execution

The concept of predicated execution is an extension of an instruction-set architecture to include instructions that execute conditionally based on the value of a boolean operator, which is called the predicate of the instruction (Hwu, 1998). Instructions with true predicate are executed normally, whereas instructions with false predicate are annulled, i.e. the instructions are not allowed to write-back their results or cause any exceptions.

#### 7.1.1 Instruction-Level Predication Support

Predicated execution is a technique that supports instruction-level parallelism by converting frequently mispredicted branches into a sequence of conditional instructions. In other words, control dependencies caused by branch instructions are converted into data dependencies. Therefore, predicated execution is also referred to as if-conversion. Of course, the advantages of predicated execu-
tion only come at the expense of additional complexity. Firstly, the design of the instruction-set architecture becomes more complex as additional predicate operands are required for instructions. Secondly, predicated execution increases program code size by up to 30% for instruction-sets supporting predication for all instructions (Connors et al., 1999).

There exists a range of architectural extensions for implementing predicated execution. We can broadly partition these techniques into two categories, which are full predication support and partial predication support. With full predication support, all instructions implemented in the ISA can be predicated using an additional source operand to specify the predicate. Partial predication support, on the other hand, provides only a limited number of instructions that can execute conditionally. Conditional move instructions are an example of this technique. While full predication provides the most flexibility and the largest potential performance increase it also requires more substantial modifications to the instruction-set than partial predication. Recent microprocessors with partial predication support are the Alpha, MIPS, PowerPC, SPARC, Pentium 4 microprocessors, which have conditional move instructions. The ARM\textsuperscript{1} has a fully predicated instruction-set.

Mahlke et al. (1995) present a study that qualitatively and quantitatively addresses the benefits of full and partial predication. Their results show that an eight-issue processor with partial predication support gives an average performance increase of 33% over a processor without predication support. Using an instruction-set with full predication support provides another 30% performance speedup.

Figure 7.1 shows the ARM assembler code for the loop-construct containing a conditional statement provided in Listing 4.2 on page 136. Both the conditional

\textsuperscript{1}The ARM (Advanced RISC Machine) architecture uses a four-bit condition code as predicate for each instruction.
### Figure 7.1: Predicated ARM assembler code for while-loop construct

And unconditional branch are eliminated from the code by using predicated instructions, which are `streq`, `strne`, and `addeq`. The elimination of the hard to predict branch instruction from the code simplifies static WCET analysis because only one execution path has to be considered. Also, the pessimism introduced by the assumption is removed that each instance of the branch associated with the conditional statement is mispredicted.

#### 7.1.2 Software Predication

Software predication transfers the concept of hardware predication support from the instruction-set to the software level. The aim of both concepts is to eliminate a branch and the branch prediction associated with it. This increases the distance between mispredictions and, therefore, the average case performance is improved because instruction scheduling is more efficient.

Listing 7.1 illustrates how the conditional statement of the `while`-loop shown in Listing 4.2 can be eliminated using software predication. Figure 7.2 shows the corresponding assembler code and control flow graph for the `while`-loop.
### 7.1. Predicated Execution

**Listing 7.1:** while-loop with software predication

```c
int_t while_loop(int_t *data, int_t n)
{
    int_t i = 0, result = 0;
    int_t cc, even, odd;

    while (i < n) {
        cc = ((data[i] % 2) == 0);
        result = result + cc;
        even = data[i] / 2;
        odd = data[i] + 1;
        cc = ~cc + 1;
        data[i++] = (even & ~cc) | (odd & cc);
    } /* end while */

    return result;
} /* end of while_loop */
```

**Figure 7.2:** while-loop construct with software predication

(a) Assembler Code

(b) Control Flow Graph
Although the control flow graph is now less complex compared with the one shown in Figure 4.6 the number of instructions in the loop body is increased due to the additional instructions introduced by software predication. Instead of four basic blocks within the original loop construct there is now only one basic block in the modified loop construct. A trade-off has to be made between the increase of code size and the benefit of reducing the number of branch instructions. In general, predication is preferable for conditional constructs that contain only a few simple statements and are frequently mispredicted, i.e. they are classified as being hard-to-predict. Such constructs, however, often do not meet the condition we have stated in Theorem 4.2. Thus, predication can improve the pessimism associated with static analysis of branch prediction by eliminating hard-to-predict branch instructions that otherwise have to be assumed to be always mispredicted.

We evaluate the effect of software predication for this small example by using the cycle-level simulator `sim-outorder`. The results are based on a four-issue out-of-order instruction pipeline without taking into account the effects of instruction and data caches, i.e. we assume a perfect memory system. The loop construct does not fulfill the condition in Theorem 4.2 such that we have to assume a misprediction each time the branch instruction associated with the conditional statement is executed. This worst-case behaviour is in fact feasible when we use an alternating sequence of even and odd values as input data.

For our example, software predication combined with the bimodal branch predictor used throughout Chapter 3 and a three cycle branch misprediction penalty, causes a performance loss of about 5.0% compared to the unmodified loop construct. We can observe in this case that the increase in the number of instructions outweighs the benefit gained from reducing the number of branch mispredictions. However, if we assume a branch misprediction penalty of at least

---

2A perfect memory system immediately provides any datum requested by the load/store unit of the microprocessor.
7.2 Unrolling of Loops

The amount of instruction-level parallelism available within a basic block is limited because branches are likely to occur in every five to seven instructions. A simple approach for increasing the amount of parallelism is to exploit parallelism among subsequent iterations of a loop. *Loop unrolling* is a commonly used technique for converting loop-level parallelism into instruction-level parallelism by issuing instructions from different loop iterations to the pipeline together. This allows the instruction scheduler to arrange the instructions in the pipeline more efficiently.

In addition to the loop unrolling performed implicitly by the instruction scheduler at run-time, the compiler or programmer can also do this statically in the source code. This can be achieved, for example, by replicating the loop body several times, and adjusting the loop header accordingly. Listing 7.2 shows a simple loop statement with a static number of iterations. Also, the loop body of an iteration is independent from that of any other iteration. The corresponding

```c
void loop(int_t *data)
{
    int_t i;

    for (i = 0; i < 4; i++) {
        data[i] = 1 << i;
    } /* end for */
} /* end of loop */
```

four cycles the situation changes and the predicated version of the loop now outperforms its original counterpart. While the performance speedup is 2.1% for a four cycle misprediction penalty, this figure increases to 22.5% if we use a misprediction penalty of eight cycles.

7.2 Unrolling of Loops
assembler code without loop unrolling is shown in Figure 7.3(a). When we use the `-funroll-loops` compiler switch of the GNU C Compiler the loop unrolling optimisation feature is activated. Figure 7.3(b) shows the generated assembler code with loop unrolling being enabled. The loop branch has been eliminated completely and, instead, the loop body is repeated four times.

```
(b1) $004001f0  addu  r3,r0,r0
     $004001f8  addiu r5,r0,1
(b2) $00400200  sllv  r2,r5,r3
     $00400208  sw    r2,0(r4)
     $00400210  addiu r4,r4,4
     $00400218  addiu r3,r3,1
     $00400220  slti  r2,r3,4
     $00400228  bne   r2,r0,b2
(b3) $00400230  jr    r31
```

(a) Without Loop Unrolling

```
(b1) $004001f0  addiu r2,r0,1
     $004001f8  sw    r2,0(r4)
     $00400200  addiu r2,r0,2
     $00400208  sw    r2,4(r4)
     $00400210  addiu r2,r0,4
     $00400218  sw    r2,8(r4)
     $00400220  addiu r2,r0,8
     $00400228  sw    r2,12(r4)
     $00400230  jr    r31
```

(b) With Loop Unrolling

Figure 7.3: Loop construct demonstrating the effect of loop unrolling

We can summarise that the benefit of using loop unrolling depends on the number of eliminated branch mispredictions. Also, the increase in code size due to replicating the loop body may have to be taken into account if memory utilisation is critical. For bimodal branch predictors, loop control branches are easy to predict and using loop unrolling is usually unjustified. However, it may well decrease the amount of pessimism associated with the static timing analysis for global history two-level predictors, e.g. GAg, by reducing the number of branch history patterns that have to be considered in the analysis.
7.3 Case Study: bladeenc Sample Program

In this section we present a case study that demonstrates how software predication discussed earlier in this chapter can be used to eliminate branches in the bladeenc sample program that has been introduced in Chapter 3. We have selected this sample program for this case study because it provides some interesting potential for coding optimisations regarding branch predictor performance. In the second part of this section we will show how a nested loop statement with an alternating conditional statement can be modified in order to reduce the number of branch mispredictions.

<table>
<thead>
<tr>
<th>Branch</th>
<th>Instances</th>
<th>Taken</th>
<th>bimod</th>
<th>GAg</th>
<th>PAp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00414520</td>
<td>3,619,008</td>
<td>99.8%</td>
<td>0.2%</td>
<td>0.8%</td>
<td>0.3%</td>
</tr>
<tr>
<td>0x00414550</td>
<td>3,619,008</td>
<td>99.6%</td>
<td>0.4%</td>
<td>0.6%</td>
<td>0.5%</td>
</tr>
<tr>
<td>0x00414580</td>
<td>3,619,008</td>
<td>99.3%</td>
<td>0.7%</td>
<td>0.9%</td>
<td>0.8%</td>
</tr>
<tr>
<td>0x004145b0</td>
<td>3,619,008</td>
<td>98.6%</td>
<td>1.6%</td>
<td>1.8%</td>
<td>1.7%</td>
</tr>
<tr>
<td>0x004145e0</td>
<td>3,619,008</td>
<td>96.6%</td>
<td>3.9%</td>
<td>4.1%</td>
<td>3.6%</td>
</tr>
<tr>
<td>0x00414610</td>
<td>3,619,008</td>
<td>91.3%</td>
<td>9.5%</td>
<td>9.8%</td>
<td>8.2%</td>
</tr>
<tr>
<td>0x00414638</td>
<td>3,619,008</td>
<td>75.0%</td>
<td>25.4%</td>
<td>23.9%</td>
<td>17.7%</td>
</tr>
<tr>
<td>Total</td>
<td>25,333,056</td>
<td>94.3%</td>
<td>6.0%</td>
<td>6.0%</td>
<td>4.7%</td>
</tr>
</tbody>
</table>

Table 7.1: Overview of most frequent branch instructions in bladeenc

Table 7.1 shows the seven branch instructions in the bladeenc sample program that have been executed most frequently. These branches account for nearly 34% of all dynamic branch instances. Listing D.1 on page 252 provides the C language source code associated with these branch instructions, which are all located in the bladTabValue function. The remaining columns of the table give the taken rate and the branch misprediction rates for bimod, GAg (highest misprediction rate), and PAp (lowest misprediction rate) predictors, respectively. Table 7.2
shows similar information for all dynamic branch instances of the bladeenc sample program. The two other sample programs bladeenc-opt and bladeenc-hyp in this table will be discussed later in this section.

The first five branch instructions shown in Table 7.1 have a taken rate of more than 95%, i.e. they fall into the SC6 static class (see Table 3.9 on page 95). Thus, the number of state transitions for these branches is very small and consequently the misprediction rate for all branch predictors is also very low. However, this behaviour is not obvious for the GAg global-history predictor because prediction takes into account the history of all branch instructions executed recently. This can be explained by the fact that the sequence of conditional statements shown in Listing D.1 is actually part of a loop construct, and therefore these mostly taken branches cause the global history pattern to be biased towards a small number of patterns.

<table>
<thead>
<tr>
<th>Branch</th>
<th>Instances</th>
<th>Taken</th>
<th>bimod</th>
<th>GAg</th>
<th>PAp</th>
</tr>
</thead>
<tbody>
<tr>
<td>bladeenc</td>
<td>75,280,109</td>
<td>72.8%</td>
<td>7.1%</td>
<td>6.4%</td>
<td>4.8%</td>
</tr>
<tr>
<td>bladeenc-opt</td>
<td>49,692,561</td>
<td>61.9%</td>
<td>7.5%</td>
<td>6.5%</td>
<td>4.8%</td>
</tr>
<tr>
<td>bladeenc-hyp</td>
<td>68,042,093</td>
<td>71.7%</td>
<td>6.0%</td>
<td>5.3%</td>
<td>4.0%</td>
</tr>
</tbody>
</table>

Table 7.2: Evaluating the effect of branch optimisation on misprediction rate

We apply now the software predication technique presented in Section 7.1.2 to our example program in order to eliminate all seven branch instructions shown in Table 7.1. The simulation results for the modified program, which is called bladeenc-opt are shown in Table 7.2. The average misprediction rate compared to the original sample program has slightly increased for all three branch predictor configurations. This is because the overall average misprediction rate of the seven branch instructions is lower than the average misprediction rate of the original bladeenc sample program. When we eliminate only the last two branches by software predication we obtain the results shown for the bladeenc-hyp sample.
7.3. CASE STUDY: BLADECENC SAMPLE PROGRAM

Listing 7.3: Nested loop example from bladeenc

```c
for (band = 0; band < 32; band++)
  for (k = 0; k < 18; k++)
    if ((band & 1) && (k & 1))
      (*sb_sample)[ch][gr+1][k][band] *= -1.0;
```

Listing 7.4: Optimised nested loop

```c
for (k = 1; k < 18; k+=2) {
  for (band = 1; band < 32; band+=2) {
    (*sb_sample)[ch][gr+1][k][band] *= -1.0;
  }
} /* end for */
} /* end for */
```

In this case the average misprediction rate is in fact reduced and is now below the original average misprediction rate. However, if we assume that the behaviour of the branch instruction depends completely on the input data provided to the program the decision as to which branches should be eliminated becomes more complicated.

Whether the average misprediction rate is improved by predication or not may depend on the input data. In the context of static WCET analysis, however, we are more interested in finding a tight upper bound on the number of mispredictions that can occur. Predication techniques are best used for branch instructions that are classified as being hard-to-predict and therefore assumed to be always mispredicted. For our example of a conditional construct, this is the case if the number of clock cycles required to execute the conditional statement is less than twice the branch misprediction penalty, as demonstrated earlier in Subsection 4.4.2 on page 136. As already mentioned, we also have to make a trade-off between the additional number of clock cycles introduced by predication and the worst-case misprediction penalty. In general, the additional number of clock cycles must be less than the branch misprediction penalty assuming that each branch instruction is mispredicted.

In our second example, we will use the concept of loop unrolling to reduce
CHAPTER 7. CODING AND COMPILATION TECHNIQUES
SUPPORTING STATIC ANALYSIS

the performance impact of hard-to-predict branches. Listing 7.3 shows the C language source code for a nested loop construct containing a conditional statement. This loop is located in the mdct_sub function of our sample program. The assignment statement will be executed if the loop variables of both the inner (loop variable \( k \)) and outer loop (loop variable \( \text{band} \)) are odd. Although this nested loop does not contribute significantly to the overall number of mispredictions in the bladeenc program, it provides an interesting example of the various improvements that are possible for loop constructs. There are two problems associated with the way the nested loop is implemented. Firstly, the conditional statement is not required at all if we change the loop counter variables such that only odd values occur. Secondly, memory access by the two-dimensional array is not linear and, therefore, the cache hit rate may be reduced.

Both problems are alleviated by the nested loop construct that is shown in Listing 7.4. The inner and outer loops are exchanged in order to improve spatial locality of memory accesses. Also, both loops iterate over odd values only, which makes the conditional statement obsolete. These loop transformations reduce the number of dynamic branches in the mdct_sub function by 66.1% and the number of mispredicted branches by 84.2%. The overall misprediction rate for this function is reduced from 34.2% to 15.9%.

7.4 Summary

This chapter has introduced predicated execution and loop unrolling in order to reduce the number of branch instructions that remain hard to predict. A short case study has shown for one of the sample programs presented earlier in this thesis that predicated execution provides a large potential for improving the performance of dynamic branch prediction by eliminating branches that are frequently mispredicted. However, it has also been shown that a trade-off between
the additional number of clock cycles introduced by predication and the penalty associated with the otherwise mispredicted branch instructions has to be made. When making this trade-off for the context of static WCET analysis, we have to take into account the worst-case branch prediction scenario rather than the average case predictor performance.
Chapter 8

Conclusions and Future Directions

The purpose of this chapter is to summarise the results of this thesis and provide suggestions for future work. Section 8.1 concludes the contributions to research in the area of static WCET analysis for dynamic branch prediction schemes and examines how the work detailed in this thesis supports the thesis proposition. Section 8.2 provides some concluding remarks on the research presented in this thesis. Last but not least, Section 8.3 identifies future areas of research that could be performed to complement the contributions of this thesis.

8.1 Summary of Achievements

In Chapter 1, the following thesis proposition was stated:

*Accurate worst-case execution time (WCET) analysis for dynamic branch predictors can be performed for a large number of branches in a program by analysis tailored by a branch classification scheme based on a theoretical model of branch prediction, and the results of the analysis then integrated with other parts of the WCET analysis.*
In the following, the contributions of each chapter and of the static analysis case studies are summarised and evaluated towards supporting the above thesis proposition. This evaluation is organised around the following set of criteria:

1. **accuracy**, **efficiency** and **scalability** of the analysis approach;

2. **relevance** of the analysis for a large number of branch instructions in a program; and

3. **modularity** in order support integration with other analysis methods.

**Evaluation Framework for Branch Predictor Behaviour**

In order to support this proposition, Chapter 3 has evaluated the branch behaviour of several sample programs by using the SimpleScalar simulator tool set. The simulation results obtained through two different classification schemes have led to the conclusion that the behaviour of the majority of branches in the selected sample programs is in fact easy-to-predict and therefore provide a promising potential for static timing analysis. However, a small number of hard-to-predict branches accounted for the majority of mispredictions in a program so we have concluded that excluding these branches from analysis does not add a significant amount of pessimism to the WCET estimate. For example, only about 2% of the static branch instructions of the bladeenc sample program exhibit worst-case predictor performance for a bimodal branch predictor. The simulation results presented in Chapter 3 have also indicated that for all four sample programs more than 50% of all branch instructions were predicted with an accuracy of at least 90%.

The analysis of the pessimism associated with static analysis of dynamic branch predictors has highlighted the fact that simply assuming a branch misprediction for every branch results in a significant level of pessimism. The amount of
pessimism caused by this assumption has varied between 71.1% and 117.8% for an out-of-order instruction pipeline and a three cycle misprediction penalty. Therefore, accurate static timing analysis of dynamic branch predictors is paramount to calculating tight WCET estimates for state-of-the-art microprocessors.

Static Analysis Approach for Bimodal Branch Predictors

Chapter 4 has established the foundation of our static analysis approach for bimodal branch predictors. Instead of using the dynamic execution behaviour of branches to classify them the classification approach has been based on the semantic context of a branch. According to our classification approach a branch instruction is classified as easy-to-predict if there exists a static branch execution pattern and this pattern can be determined at compile-time from the semantic context of the branch by means of static analysis. Otherwise, we classify the branch instruction as hard-to-predict in terms of static WCET analysis. Using this classification approach, an upper bound on the number of branch mispredictions has been derived for various control statements of the C programming language, like loop constructs and conditional statements. For the latter, the condition originally stated by Colin and Puaut (1999) has been corrected.

The branch classification approach presented in Chapter 4 has been published previously in Bate and Reutemann (2004). In a later work, Burguière and Rochange (2005) extended our previous work in order to differentiate the execution time for the taken and the not-taken path. They have also considered more general loop nests, where the internal loop can have variable iteration counts.

Integrated with Instruction Pipeline Analysis

Chapter 5 has elaborated on the integration of individual WCET analysis methods and demonstrated that this integration is not straightforward due to the
interaction between the analyses. This often leads to large increases in computational complexity, the introduction of unnecessary pessimism, or even to unsafe WCET estimates.

However, based on the WCET analysis approach presented in Chapter 4 of this thesis we have shown that branch predictors can be integrated with instruction pipeline analysis and the WCET be estimated using an ILP-based calculation method. This is achieved by first calculating the number of branch mispredictions and then representing these as constraints. Taking this approach results in significantly fewer constraints than for other approaches estimating misprediction numbers as part of the ILP problem.

**Static Analysis of Two-Level Branch Predictors using Global History**

Chapter 6 has extended the static analysis method to two-level branch predictors using global history. For such branch predictors it is no longer possible to associate a branch instruction with a particular entry in the predictor table. While static analysis for bimodal and local history predictors can be restricted to the branch instruction being analysed, the scope of analysis has to be extended for global-history predictors in order to take into account branch behaviour across multiple branch instructions. This complicates the analysis and may also increase the pessimism because the global state of the predictor may not always be known at compile-time.

Based on our work published previously in Bate and Reutemann (2004), we have explained the effects observed by Engblom (2002, 2003), which he called *inversions*, for the Intel Pentium III, AMD Athlon, and UltraSPARC III microprocessors. For the AMD Athlon microprocessor, the total execution time rises significantly between the ninth and tenth iteration of the inner loop. In order to explain this behaviour, we have developed a theoretical model based on branch patterns for global history branch predictors. Assuming \( n \) iterations of a inner
loop and a global branch history register size of \( k \) bits, the maximum number of branch mispredictions can be expected for \((n - k) = 2\), i.e. the number of iterations exceeds the branch history register size by two.

The experimental results supporting our theoretical model have been obtained by SimpleScalar using a branch predictor configuration that resembles the one of the AMD Athlon microprocessor. In addition, the results have been confirmed by the author on a real Athlon microprocessor using its performance counters.

**Coding and Compilation Techniques Supporting Static Analysis**

Chapter 7 has discussed techniques for reducing the pessimism associated with the static analysis of hard-to-predict branches. Predicated execution provides a large potential for eliminating branches that are frequently mispredicted. The impact of software predication on the overall number of mispredictions has been analysed by a case study using a modified version of the `bladeenc` sample program. In this case study, we have eliminated the seven most frequently executed branch instructions, which accounted for nearly 34% of all dynamic branch instances. The simulation results have shown that it is important to make a trade-off between the additional number of clock cycles introduced by predication and the misprediction penalty.

**Static Analysis Case Studies**

The case study examples presented in this thesis have demonstrated that local-history (such as bimodal predictors) and global-history branch predictors are not equally well suited for static WCET analysis, primarily in terms of efficiency and scalability. As it is generally the case for embedded real-time systems simple techniques are favourable compared to more complex ones having a performance gain in the average case. As far as static WCET analysis is concerned, developers
should use bimodal branch predictors instead of more complex two-level or combined predictor schemes. This is because the slight performance gain achieved by two-level or combined predictors does not justify the significant additional complexity introduced to WCET analysis that is required to accurately model such predictors. It may also be possible that the performance gain is even outweighed by the additional pessimism caused by the wider scope of analysis necessary to model global branch history effects.

8.2 Concluding Remarks

We have shown in the course of this thesis that accurate bounds on the number of mispredictions can be obtained for bimodal predictors using a branch classification scheme. The relevance of this branch classification scheme has been demonstrated by an evaluation framework.

The performance benefit achieved by bimodal branch predictors exceeds the pessimism introduced by the static analysis approach. Therefore, using bimodal branch predictors in real-time systems can be considered feasible. However, this cannot be said for more complex two-level predictors where the additional pessimism caused by the wider scope of analysis usually outweighs the slight performance gain over bimodal predictors. In summary we can conclude that the objectives of this work were met.

A particular difficulty with static WCET analysis for dynamic branch predictors arises from the fact that for many recent microprocessors the exact configuration of the branch prediction unit is not provided by their manufacturers. This is because today branch prediction is considered a leading-edge feature for achieving high processing performance figures – particularly for microprocessors using very long instruction pipelines. It has been predicted already ten years ago that branch prediction techniques will become a crucial factor for improving the
8.3 Future Directions

This thesis has identified a number of areas in which further research work could be performed. These areas are briefly outlined in the following subsections, which are ordered in decreasing order of priority (according to the opinion of the author).

8.3.1 Integration With Cache Analysis

The most obvious avenue for further work is integrating the analysis approach for dynamic branch predictors with static timing analysis for instruction pipelines and caches. While Chapter 5 has shown that branch prediction analysis can be integrated with instruction pipeline analysis, a potential area for future work would be to demonstrate the feasibility of our approach through a case study for a real microprocessor architecture featuring a more sophisticated instruction pipeline with out-of-order and speculative execution.

Integrating various low-level analysis techniques remains challenging. This is because the modularity of the individual analyses may suffer due to the inter-dependencies among the different micro-architectural features. For example, a mispredicted branch causes instructions being fetched from the wrong path and may therefore pollute the contents of the instruction cache causing additional cache misses. Addressing such aspects requires that the scope of low-level timing
analysis techniques is extended to also include global aspects involving multiple models and their interaction with each other.

The integration into an overall analysis framework, for example as depicted earlier in Figure 1.2 on page 29, would also require that the approach can take into account different branch predictor configurations. It would be convenient to specify the configuration of a branch predictor by means of a simple description notation, which could be based on the command line format required by the SimpleScalar tools.

### 8.3.2 Branch Interference

Branch interference complicates static timing analysis for dynamic branch predictors because the scope of the analysis has to be extended in order to cover the behaviour of branches competing for the same predictor entry.

This approach requires that we widen the scope of static WCET analysis to cover global effects across multiple branch instructions in order to identify all branch instructions that may suffer from destructive interference and, therefore, actually cause additional mispredictions. For the analysis approach presented in Chapter 4, for example, this would mean that the assumptions regarding worst-case initial branch predictor states are no longer necessarily valid in the case of branch interference.

Alternatively, we can try to eliminate branch interference, which is quite simple for per-address branch predictors because the branch addresses are known statically. However, this approach would require that the linker is modified in order to generate or transform the program code such that no two branch instructions share the same predictor entry. The advantage of this approach is not only a less complex analysis model but also that branch interference is removed from the code and thus the software performance is improved in general.
8.3.3 Modelling Indirect Branches

Assess the impact of prediction of indirect branches on static WCET analysis. Most prediction schemes for indirect branches use a BTB or BTAC to obtain the target address of the branch. This problem is similar to data caches and, therefore, adaptation of existing analysis methods for caches should be possible in order to account for prediction of branch target addresses.
Appendix A

Mispredictions per Static Class
Figure A.1: Box-plots of mispredictions per static class (gcc-opt)
Figure A.2: Box-plots of mispredictions per static class (gcc)
APPENDIX A. MISPREDICTIONS PER STATIC CLASS

Figure A.3: Box-plots of mispredictions per static class (bzip2)
Appendix B

Mispredictions per Branch Type
Figure B.1: Box-plots of mispredictions per branch type (gcc-opt)
## Appendix B: Mispredictions Per Branch Type

### Figure B.2: Box-plots of mispredictions per branch type (gc c)

<table>
<thead>
<tr>
<th>Branch Type</th>
<th>Misprediction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.0</td>
</tr>
<tr>
<td>C</td>
<td>0.2</td>
</tr>
<tr>
<td>L</td>
<td>0.4</td>
</tr>
<tr>
<td>N</td>
<td>0.6</td>
</tr>
<tr>
<td>S</td>
<td>0.8</td>
</tr>
<tr>
<td>T</td>
<td>1.0</td>
</tr>
</tbody>
</table>

![Box-plots of mispredictions per branch type (gc c)](image-url)
Figure B.3: Box-plots of mispredictions per branch type (bz ip2)
Appendix C

Listing for Timing Analysis Case Study

Listing C.1: Example program for timing analysis case study

```c
#include <stdio.h>
#include <stdlib.h>

#define SAMPLE_SIZE 20
#define NUM_SAMPLES 4

typedef unsigned long ulong_t;

typedef struct result_struct {
    ulong_t num_even, sum, qsum;
} result_t;

ulong_t sample_data[NUM_SAMPLES][SAMPLE_SIZE] = {
    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19 },
    { 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20 },
    { 2, 4, 6, 8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40 },
    { 1, 3, 5, 7, 9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39 }
};

c char *sample_name[NUM_SAMPLES] = {
    "alternating_even_first", "alternating_odd_first",
    "all_even", "all_odd"
};

result_t while_loop(ulong_t *data)
{
    result_t res;
    ulong_t i = 0;
```
APPENDIX C. LISTING FOR TIMING ANALYSIS CASE STUDY

```c
res.num_even = 0;
res.sum = 0;
res.qsum = 0;

while (i < SAMPLE_SIZE) {
    if ((data[i] % 2) == 0) {
        data[i] = data[i] / 2;
        res.sum += data[i];
        res.qsum += data[i] * data[i];
        res.num_even++;
    } else {
        data[i]++;
    }
    i++;
}

return res;
```

```c
int main(int argc, char *argv[]) { 
    result_t res;
    int i, sample, exit_value = 0;

    if (argc < 2) {
        fprintf(stderr, "ERROR:␣no␣sample␣id(s)␣provided.
" );
        exit_value = 1;
    } else {
        for (i = 1; i < argc; i++) {
            sample = atoi(argv[i]);
            if ((sample > 0) && (sample <= NUM_SAMPLES)) {
                printf("\nSAMPLE␣%d:␣%s
", sample, sample_name[sample - 1]);
                res = while_loop(sample_data[sample - 1]);
                printf("num_even␣=␣%ld
", res.num_even);
                printf("sum␣␣␣␣␣␣=␣%ld
", res.sum);
                printf("qsum␣␣␣␣␣=␣%ld
", res.qsum);
            } else {
                fprintf(stderr, "ERROR:␣id␣%d␣not␣within␣range␣1..%d
", sample, NUM_SAMPLES);
                exit_value = 1;
                break;
            }
        }
        exit(exit_value);
    }
}
```

246
Listing C.2: Conditional construct embedded within loop – NPB

typedef struct result_struct {
    ulong_t num_even;
    ulong_t sum;
    ulong_t qsum;
    ulong_t csum;
} result_t;

result_t cond_loop_npb(ulong_t *data,
    ulong_t *out,
    ulong_t num_samples)
{
    result_t res;
    ulong_t i;
    res.num_even = 0;
    res.sum = 0;
    res.qsum = 0;

    for (i=0; i<num_samples; i++) {
        if ((data[i] % 2) == 0) {
            out[i] = data[i] / 2;
            res.sum += data[i];
            res.qsum += data[i] * data[i];
            res.csum += data[i] * data[i] * data[i];
            res.num_even++;
        } else {
            out[i]++;
        }
    }
    return res;
} /* end of cond_loop_npb */

/* end_of_cond_loop_npb */
### APPENDIX C. LISTING FOR TIMING ANALYSIS CASE STUDY

**b1:**
- $004001f0$ addiu r29, r29, -16
- $004001f8$ addu r7, r0, r4
- $00400200$ addu r6, r0, r0
- $00400208$ sw r0, 0(r29)
- $00400210$ sw r0, 4(r29)
- $00400218$ sw r0, 8(r29)

**b2:**
- $00400220$ lw r3, 0(r5)
- $00400228$ andi r2, r3, 1
- $00400230$ bne r2, r0, b4

**b3:**
- $00400238$ srl r3, r3, 0x1
- $00400240$ sw r3, 0(r5)
- $00400248$ lw r2, 4(r29)
- $00400250$ addu r2, r2, r3
- $00400258$ sw r2, 4(r29)
- $00400260$ lw r2, 0(r5)
- $00400268$ mult r2, r2
- $00400270$ lw r4, 8(r29)
- $00400278$ lw r2, 0(r29)
- $00400280$ mflo r3
- $00400288$ addiu r2, r2, 1
- $00400290$ addu r3, r3, r4
- $00400298$ sw r2, 0(r29)
- $004002a0$ sw r3, 8(r29)
- $004002a8$ j b5

**b4:**
- $004002b0$ addiu r2, r3, 1
- $004002b8$ sw r2, 0(r5)

**b5:**
- $004002c0$ addiu r5, r5, 4
- $004002c8$ addiu r6, r6, 1
- $004002d0$ sltiu r2, r6, 20
- $004002d8$ bne r2, r0, b2

---

Figure C.1: Assembler code for timing analysis case study
APPENDIX C. LISTING FOR TIMING ANALYSIS CASE STUDY

Figure C.2: Assembler code for timing analysis case study (cont.)

<table>
<thead>
<tr>
<th>b6:</th>
<th>address</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$004002e0</td>
<td>lw</td>
<td>r2,0(r29)</td>
</tr>
<tr>
<td>$004002e8</td>
<td>lw</td>
<td>r3,4(r29)</td>
</tr>
<tr>
<td>$004002f0</td>
<td>lw</td>
<td>r4,8(r29)</td>
</tr>
<tr>
<td>$004002f8</td>
<td>sw</td>
<td>r2,0(r7)</td>
</tr>
<tr>
<td>$00400300</td>
<td>sw</td>
<td>r3,4(r7)</td>
</tr>
<tr>
<td>$00400308</td>
<td>sw</td>
<td>r4,8(r7)</td>
</tr>
<tr>
<td>$00400310</td>
<td>addu</td>
<td>r2,r0,r7</td>
</tr>
<tr>
<td>$00400318</td>
<td>addiu</td>
<td>r29,r29,16</td>
</tr>
<tr>
<td>$00400320</td>
<td>jr</td>
<td>r31</td>
</tr>
</tbody>
</table>
Appendix D

Listings for bladeenc Case Study
Listing D.1: Sequence of conditional statements from bladeenc

```c
if (in2 >= *pTabEntry++) {
    pTabEntry += 64-1;
    retVal += 64;
}

if (in2 >= *pTabEntry++) {
    pTabEntry += 32-1;
    retVal += 32;
}

if (in2 >= *pTabEntry++) {
    pTabEntry += 16-1;
    retVal += 16;
}

if (in2 >= *pTabEntry++) {
    pTabEntry += 8-1;
    retVal += 8;
}

if (in2 >= *pTabEntry++) {
    pTabEntry += 4-1;
    retVal += 4;
}

if (in2 >= *pTabEntry++) {
    pTabEntry += 2-1;
    retVal += 2;
}

if (in2 >= *pTabEntry) {
    retVal += 1;
}
```
Listing D.2: Conditional statements eliminated by software predication

1. \( \text{cc} = (\text{in2} >= \ast \text{pTabEntry}++); \)
   \( \text{cc} = \neg \text{cc} + 1; \)
   \( \text{pTabEntry} += (63 \& \text{cc}); \)
   \( \text{retVal} += (64 \& \text{cc}); \)

2. \( \text{cc} = (\text{in2} >= \ast \text{pTabEntry}++); \)
   \( \text{cc} = \neg \text{cc} + 1; \)
   \( \text{pTabEntry} += (31 \& \text{cc}); \)
   \( \text{retVal} += (32 \& \text{cc}); \)

3. \( \text{cc} = (\text{in2} >= \ast \text{pTabEntry}++); \)
   \( \text{cc} = \neg \text{cc} + 1; \)
   \( \text{pTabEntry} += (15 \& \text{cc}); \)
   \( \text{retVal} += (16 \& \text{cc}); \)

4. \( \text{cc} = (\text{in2} >= \ast \text{pTabEntry}++); \)
   \( \text{cc} = \neg \text{cc} + 1; \)
   \( \text{pTabEntry} += (7 \& \text{cc}); \)
   \( \text{retVal} += (8 \& \text{cc}); \)

5. \( \text{cc} = (\text{in2} >= \ast \text{pTabEntry}++); \)
   \( \text{cc} = \neg \text{cc} + 1; \)
   \( \text{pTabEntry} += (3 \& \text{cc}); \)
   \( \text{retVal} += (4 \& \text{cc}); \)

6. \( \text{cc} = (\text{in2} >= \ast \text{pTabEntry}++); \)
   \( \text{cc} = \neg \text{cc} + 1; \)
   \( \text{pTabEntry} += \text{cc}; \)
   \( \text{cc} = \neg \text{cc} + 1; \)
   \( \text{retVal} += (2 \& \text{cc}); \)

7. \( \text{cc} = (\text{in2} >= \ast \text{pTabEntry}); \)
   \( \text{retVal} += \text{cc}; \)
Bibliography


Bate, I. and Reutemann, R. (2004). Worst-Case Execution Time Analysis for


ILP Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 138–149, Santa Margherita Ligure, Italy.


International Symposium on Computer Architecture (ISCA), pages 257–266, San Diego, California, USA.

