UNIVERSIDADE FEDERAL DE SANTA CATARINA - UFSC
PROGRAMA DE PÓS - GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS - PPGEAS

Andreu Carminati

CONTRIBUTIONS TO WORST-CASE EXECUTION TIME REDUCTION USING COMPILATION TECHNIQUES

Florianópolis
2017

Andreu Carminati

CONTRIBUTIONS TO WORST-CASE EXECUTION TIME REDUCTION USING COMPILATION TECHNIQUES

A Thesis submitted to the Department of Automation and Systems Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Automation and Systems Engineering.
Supervisor: Rômulo Silva de Oliveira

Florianópolis
2017

,
 Contributions to Worst-Case Execution Time Reduction Using Compilation Techniques : / Andreu Carminati; orientador, Rômulo Silva de Oliveira. - Florianópolis, Brazil 2017.
 246 p.

- Universidade Federal de Santa Catarina - UFSC, Centro Tecnológico - CTC. Programa de Pós - Graduação em Engenharia de Automação e Sistemas - PPGEAS.

Inclui Referências

  I. , .   II. Universidade Federal de Santa Catarina - UFSC. Programa de Pós - Graduação em Engenharia de Automação e Sistemas - PPGEAS. II. Contributions to Worst-Case Execution Time Reduction Using Compilation Techniques.

CONTRIBUTIONS TO WORST-CASE EXECUTION TIME REDUCTION USING COMPILATION TECHNIQUES

Andreu Carminati

This Thesis is hereby approved and recommend for acceptance in partial fulfillment of the requirements for the degree of "Doctor of philosophy in Automation and Systems Engineering."

October 26th, 2017.

Prof. Rômulo Silva de Oliveira, Dr.
Supervisor

Prof. Daniel Coutinho, Dr.
Coordinator of the Automation and Systems Engineering Postgraduate Program

Examining Committee:

Prof. Rômulo Silva de Oliveira, Dr. - UFSC
Chair

Prof. Rodolfo Jardim de Azevedo, Dr. - UNICAMP

Prof. Luiz Cláudio Villar dos Santos, Dr. - UFSC

Prof. Joni da Silva Fraga, Dr. - UFSC

Our virtues and our failings are inseparable, like force and matter. When they separate, man is no more.
Nikola Tesla

ACKNOWLEDGEMENTS

I thank to my advisor professor Rômulo Silva de Oliveira for his readiness, dedication, and the remarkable suggestions that helped me to stay in the right path of the work.

I thank my family for their trust and support during my academic life. In particular I thank my parents allowing access to high end education. A special thank to my wife Kátia Morgana who encouraged and supported me for all these years.

I also thank to laboratory colleagues and friends for sharing ideas and improvements. A special thank to my friend Renan Starke who helped to make this work possible with the processor implementation, long debugging sessions and priceless discussions.

Finally thanks to CAPES for the financial support.

ABSTRACT

A wide range of systems are distinct from the general purpose computing systems due to the need of satisfying rigorous timing requirements, often under the constraint of available resources, they are generally called real-time systems. The development of a predictable system is concerned with the challenges of building systems whose time requirements can be guaranteed *a priori*. Although, these challenges become even greater when using processors' architectural features for performance increase, as caches and pipelines, which introduce a high degree of uncertainty, making difficult to provide any kind of guarantee. Parallel to this, there are the tools needed to develop and execute an application, such as languages, compilers, runtime support, communication systems and scheduling, which may further make difficult the assertion of guarantees. In these systems, the results of computations must be generated at the right time and faults of temporal nature can result in catastrophic consequences both in the economic sense as in human lives. These systems are present in countless applications, such as in industrial plants, aviation, and the complexity of them imposes serious restrictions on the hardware that can be used. To provide timing guarantees, we must know the worst-case execution time for each tasks of the system. In a general purpose architecture aimed at the average case, the execution time of a program or task can be so great in the worst case that invalidates the design constraints, or even be impossible to be calculated or estimated with a reasonable effort. In this thesis, we integrate compilation with WCET calculation. A compiler can provide relevant data to facilitate the process of WCET estimation. To improve this process, we also use an architecture whose purpose is to conciliate performance with determinism. Considering compilation and WCET integration we present the following contributions: (1) a different way to perform loop unrolling on data-dependent loops using code predication targeting WCET reduction, because existing techniques only consider loops with fixed execution counts. (2) considering static branch predication techniques, we show that a very small gain or even none can be obtained with new optimization techniques targeted to worst-case exe-

cution time reduction. To achieve this objective, we compare several techniques against the perfect branch predictor. (3) the difference between the WCET of a task and its actual execution time is called gain time. We propose a technique that finds specific points of a program (called gain points), where there will be an amount of statically estimated gain time in the case that path is taken by the execution.
Keywords: Real time systems, Compilation, Worst-Case Execution Time (WCET) Analysis.

RESUMO EXPANDIDO

CONTRIBUIÇÕES PARA A REDUÇÃO DO PIOR TEMPO DE COMPUTAÇÃO UTILIZANDO TÉCNICAS DE COMPILAÇÃO

Palavras-chave: Sistemas de tempo real, compilação, análise de pior tempo de computação (WCET – Worst-case Execution Time)

Introdução

Uma grande gama de sistemas se distinguem dos sistemas de computação de propósito geral pela necessidade de satisfação de requisitos de temporização rigorosos. O desenvolvimento de um sistema previsível preocupa-se com os desafios de construção de sistemas cujos requisitos temporais possam ser garantidos *a priori*. Estes desafios tornam-se ainda maiores quando se utiliza recursos arquiteturais para aumento de performance, como *caches* e *pipelines*, os quais introduzem um alto grau de incertezas, tornando difícil o provimento de qualquer tipo de garantia. Paralelamente a isto, existem as ferramentas necessárias ao desenvolvimento e execução da aplicação, como linguagens, compiladores, *runtime* de execução, sistemas de comunicação e escalonamento, os quais podem dificultar ainda mais a asserção de garantias.

Nestes sistemas, os resultados das computações devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto. As falhas de natureza temporal nestes sistemas são, em alguns casos, consideradas críticas no que diz respeito às suas consequências. Nos sistemas tempo real críticos (*hard real-time*) o não atendimento de um requisito temporal pode resultar em consequências catastróficas tanto no sentido econômico quanto em vidas humanas. Quando os requisitos temporais não são críticos (*soft real-time*) eles apenas descrevem o comportamento desejado. O não atendimento de tais requisitos reduz a utilidade da aplicação mas não a elimina completamente nem resulta em consequências catastróficas.

Estes sistemas estão presentes em diversas aplicações, como em plantas industriais, aviação e eletrônica automotiva, telecomunicações e sistemas espaciais. Em várias destas aplicações, a complexidade dos sistemas de software impõe sérias restrições quanto ao hardware que poderá ser utilizado. Este deverá ter capacidade suficiente para sustentar a aplicação em questão, além de poder estar submetido a restrições não funcionais do projeto, como custo e eficiência energética.

Arquiteturas modernas e de propósito geral possuem como premissa básica aquela que diz que os programas devem executar o mais rápido possível na maioria das vezes. Este tempo médio é geralmente chamado de ACET - *Average-case Execution Time*. Entretanto, em alguns casos, o tempo de uma execução de uma aplicação poderá ser grande em relação ao caso médio, mas ainda estará amortizado entre as diversas execuções do programa. Esta priorização de caso médio impõe certas problemáticas quanto à utilização deste tipo de arquitetura em sistemas de tempo real. Tais sistemas podem exigir garantias de tempo de execução difíceis de serem obtidas ou muitas vezes inviáveis. Estas garantias exigem o conhecimento do pior tempo de execução de um programa ou tarefa em um determinado processador, o qual geralmente é chamado de WCET - *Worst-case Execution Time*. Em uma arquitetura de propósito geral que vise o caso médio, o tempo de execução no pior caso de um programa ou tarefa pode ser tão grande que inviabilize as restrições de projeto, ou mesmo ser impossível de ser estimado.

Atualmente, existem vertentes acadêmicas que sugerem a utilização de processadores e arquiteturas voltadas para aplicações de tempo real. Tais arquiteturas adotam características de hardware que tornam as análises referentes à obtenção de WCET mais simples e rápidas.

Uma característica importante é que o desempenho em arquiteturas específicas, como as voltadas para tempo real, pode estar intimamente relacionado ao compilador e as técnicas de compilação empregadas, como exploração estática de paralelismo. Dada a possibilidade de ser obter o WCET de programas para uma arquitetura específica, pode-se utilizar estas informações no processo de otimização incremental dos mesmos. Estas otimizações visam a redução do WCET, visto que abordagens tradicionais de transformação de código feitas por compilador podem até mesmo aumentar o WCET de um programa.

Objetivos

O objetivo deste trabalho é contribuir com aspectos relacionados à compilação para sistemas de tempo real, cujo objetivo primário seja a redução de WCET ou melhoria de aspectos relacionados à escalonabilidade. A tese a ser demonstrada é que o íntimo acoplamento de um compilador com um analisador WCET pode beneficiar tanto a análise quanto a síntese de um programa executável ou sistema completo para uma arquitetura determinista. A utilização de uma arquitetura determinista representa uma característica importante deste trabalho, bem como o desenvolvimento do respectivo analisador WCET.

Dentre os elementos relacionados ao compilador essenciais para a redução do WCET, pode-se citar:

- Mecanismos para o cálculo de WCET de programas em processo de compilação. Isto implica acoplamento do compilador com o analisador desenvolvido.
- Identificação de potenciais pontos a serem beneficiados por otimizações. Este processo envolve interpretação dos resultados do analisador.
- Descarte de alterações de códigos que aumentem o WCET. Novamente, decisões deverão ser tomadas com base em análises sucessivas.

Além dos elementos relacionados, podemos destacar a eficiência do processo. O uso de uma arquitetura projetada para aplicações em tempo real permite o uso de um analisador muito mais rápido e preciso, que visa trazer eficiência ao processo. Embora a arquitetura se baseie em um ISA comercial, não existe compilador livre disponível para esta, então, a implementação de um gerador de código inteiramente funcional fez-se necessária como requisito para realização do trabalho de tese.

Entre os elementos considerados como foco desta tese, têm-se:

- Técnicas de *loop unrolling*: Laços são frequentemente bons candidatos-alvo para otimizações de compilação para extrair o desempenho em processadores modernos. Algumas técnicas foram propostas na literatura para alcançar a redução do WCET usando o *loop unrolling*, como em (ZHAO et al., 2006) e (LOKUCIEJEWSKI; MARWEDEL, 2010).

Nestes trabalhos, apenas os laços com contagens de execução fixas são considerados.

- Previsão estática de desvios: Previsores de desvio são utilizados para aumentar o desempenho de programas em arquiteturas modernas. Previsores estáticos podem depender do compilador para definir o comportamento de cada desvio condicional. Esse comportamento é então adotado pelo processador para toda a execução do programa. O uso da previsão estática de desvio como mecanismo para redução do tempo de execução de pior caso é uma alternativa conhecida e foi primeiramente proposta por (BODIN; PUAUT, 2005) e (BURGUIERE et al., 2005).
- Identificação de tempo ganho em programas: Tempo ganho (ou *gain time*) (AUDSLEY et al., 1994) (AVILA et al., 2003) (HU et al., 2002) (HU et al., 2003) é a diferença entre o WCET de uma tarefa e o tempo de execução real. Uma abordagem comum é identificar o *gain time* em tempo de execução comparando o tempo de execução real (medido) com o WCET calculado estaticamente. A identificação do tempo de ganho precoce é útil para aumentar a utilização do sistema em tempo de execução e para economizar energia do sistema, por exemplo.

Alcançar a redução do pior tempo de computação em tarefas que compõem um sistema de tempo real é importante pois permite que recursos computacionais não sejam desperdiçados, impactando diretamente no custo. Outra importância para tal redução é a aceitação de tarefas do tipo *soft real-time*, pois quanto menor o WCET das tarefas do tipo *hard*, mais tempo de processador pode ser alocado para este tipo de tarefa.

Contribuições

As contribuições desta tese para o estado da arte são:

1. A proposição de uma maneira diferente de executar o *loop unrolling* sobre laços cujas execuções são dependentes de dados usando a predicação de código visando redução de WCET, porque as técnicas existentes consideram apenas laços com contagens de execução fixas. A técnica proposta também foi combinada com abordagens de *loop unrolling*

existentes. Os resultados mostraram que esta combinação pode produzir agressivas reduções de WCET quando comparadas com o código original.

2. Em relação às técnicas de predição estática de desvios, são mostrados que somente ganhos pequenos ou mesmo nenhum ganho pode ser obtido com novas técnicas de otimização direcionadas para a redução do tempo de execução do pior caso. Para alcançar esse objetivo, foram comparadas várias técnicas contra o previsor de desvio perfeito. Este previsor permite estimar a redução máxima de WCET que pode ser obtida com abordagens estáticas. Além da técnica clássica da literatura, foi incluída na comparação uma nova técnica centrada em WCET que atua como uma abordagem de força bruta para aproximar os resultados do preditor perfeito. A comparação também inclui técnicas de compilação não diretamente orientadas para redução de WCET. Como resultado, são mostradas que as técnicas consideradas nesta tese estão próximas do resultado ótimo obtido pelo previsor perfeito. Também é mostrado que a técnica proposta produz resultados ligeiramente melhores do que as demais técnicas. Como contribuição secundária, é mostrado que as técnicas inconscientes de WCET também podem ser usadas em ambientes em tempo real porque apresentam bons resultados e baixa complexidade. As técnicas de previsão foram avaliadas usando um conjunto de exemplos dos *benchmarks* para WCET de Mälardalen.

3. Um problema do WCET é que ele é relativo a um único caminho de execução, especificamente o caminho de execução do pior caso (WCEP). Quando uma aplicação em tempo real executa sobre um caminho diferente do WCEP, seu tempo de execução será provavelmente menor do que o WCET. A diferença entre o WCET de uma tarefa e seu tempo de execução real é chamado de tempo ganho. Neste trabalho, é proposta uma técnica que encontra pontos específicos de um programa (chamados pontos de ganho), onde haverá uma quantidade de tempo ganho estimado estaticamente no caso de esse caminho ser tomado pela execução. Como estudo de caso, é apresentado o tempo ganho obtido pela aplicação estratégia proposta a um *benchmark* da série de *benchmarks*

para WCET de Mälardalen. Para o *benchmark* selecionado, foram identificados vários pontos de ganho e alguns deles com uma quantidade significativa de tempo ganho detectado estaticamente.

Conclusão

Sistemas de tempo real estão presentes em diversos segmentos da indústria, desde sistemas aviônicos a eletrônica automotiva, passando por sistemas industriais. No passado, tais sistemas eram bastante simples, considerando a demanda por recursos computacionais e interdependência entre tarefas. Porém hoje o cenário é outro: têm-se aplicações com altíssimo nível de complexidade, por vezes geradas sem intervenção humana a partir de modelos formais. Cada tarefa componente destas aplicações possui seu próprio prazo e por vezes depende de resultados provenientes de outras tarefas (possivelmente através de uma rede), levando a necessidade de estimativa também de prazos fim-a-fim.

Levantado o cenário anterior, percebe-se que processadores simples, como microcontroladores, não são capazes de atender aplicações de tempo real como atendiam no passado. Neste caso, torna-se necessária a utilização de processadores com maior capacidade computacional, com mecanismos de aumento desempenho, como *pipelines*, *caches* e execução especulativa. O problema com estes mecanismos é a dificuldade de cálculo do pior caso no tempo de computação, devido a fatores como anomalias temporais. Entretanto, algumas vertentes da literatura sugerem o uso de arquiteturas voltadas para tempo-real, ou seja, deterministas.

Neste trabalho, foi objetivada a geração e otimização de código para uma arquitetura determinista mas com mecanismos de aumento de performance. O objetivo primário foi a redução de WCET de programas, bem como o levantamento de alguns parâmetros úteis no projeto de um sistema de tempo real. A redução de WCET importante para não sobre-dimensionar sistemas, não desperdiçando assim, recursos computacionais. A utilização de uma arquitetura determinista aliada a redução de WCET induz a sistemas bem dimensionados em termos de recursos.

Usando técnicas como *loop unrolling* usando predicação de código e previsão estática de desvios, foi possível reduzir o pior caso no tempo de computação de tarefas. A caracterização de tempo ganho, do ponto de vista puramente estático, também pôde ser alcançada neste trabalho.

LIST OF FIGURES

Figure 1 – Example distribution of execution times of a hypothetical program in a general purpose architecture (WILHELM et al., 2008) 35
Figure 2 – Common pass structure of a compiler 47
Figure 3 – Control flow graph of the example 55
Figure 4 – Precedence DAG for the example 57
Figure 5 – Example showing the movement of instructions to above lateral entry 59
Figure 6 – Example showing the movement of instructions to below lateral entry 60
Figure 7 – Superblock formation example 63
Figure 8 – Example anomaly caused by speculation (WILHELM et al., 2008) 68
Figure 9 – Example of scheduling anomaly(WILHELM et al., 2008) 69
Figure 10 – Techniques to WCET search/calculation, obtained from (ERMEDAHL, 2003) 77
Figure 11 – Example showing a WCEP switch after the application of an optimization 80
Figure 12 – Scenarios of invariances and possible WCEP switch . . 82
Figure 13 – Example of loop unswitching. (a) before, (b) after . . . 89
Figure 14 – Example illustrating the superblock formation 92
Figure 15 – Example illustrating the path duplication technique. . . 93
Figure 16 – Example of loop unrolling optimization and superblock formation 94
Figure 17 – Example of a sequence of basic blocks with and without branch preload. 112
Figure 18 – Data path of the processor 113
Figure 19 – Possible use of Clang with LLVM and linking with Binutils 116
Figure 20 – Use of Clang with LLVM and finalization with a custom linker 118
Figure 21 – Packaging instructions considering anti-dependencies . 121

Figure 22 – Instructions alignment in a basic block 122
Figure 23 – Example illustrating the formation of a complete CFG (or ICFG) 127
Figure 24 – A C program and its respective CFG 130
Figure 25 – Cache abstract reachable state example 134
Figure 26 – CONFLICT classification example. 137
Figure 27 – Example of timing composition of two successive basic blocks 140
Figure 28 – IPET result 146
Figure 29 – Multigraph of the example 148
Figure 30 – IPET result considering cache 149
Figure 31 – Diagram representing the tools that compose the infrastructure used 153
Figure 32 – Control flow graph of Code 6.1. 159
Figure 33 – Control flow graph of Code 6.2. 160
Figure 34 – Control flow graph of Code 6.3. 163
Figure 35 – Control flow graph representing an *If-Conversion* of the code from Code 6.3. 164
Figure 36 – Graphic comparison of WCET reduction of Table 11. . 175
Figure 37 – Example of an if-then-else sentence. 187
Figure 38 – Decision tree that describes when penalties occur (or not) in the case of static branch prediction. 188
Figure 39 – Control flow graph of Code 7.1. 190
Figure 40 – Example of predictions of the branches of Figure 39. . . 191
Figure 41 – Bar chart representing the WCET improvement for each technique in relation to the *all-mispredicted* case. . . . 204
Figure 42 – Graphic comparison of optimalities of Table 15. 205
Figure 43 – Control flow graph of Code 7.2 showing the selected prediction of each approach. 206
Figure 44 – Control flow graph with a WCEP marked in black. . . . 210
Figure 45 – Control flow graph with a selected basic block (basic block 2, with a thicker border). 211
Figure 46 – Control flow graph with the execution path passing over the selected block. 212

Figure 47 – Reduced control flow graph produced by Algorithm 12. 213
Figure 48 – Gain times and WCET until the gain points 216

LIST OF TABLES

Table 1 – Instructions per basic block and resource demand for the example . . . 56
Table 2 – Scheduling obtained for the example . . . 58
Table 3 – Summary of techniques for WCET reduction . . . 108
Table 4 – Execution units in each pipeline . . . 113
Table 5 – Cycles for different types of instructions. . . . 138
Table 6 – Hazards between operations. . . . 139
Table 7 – Penalties for different types of flow transfers. . . . 140
Table 8 – Used benchmarks from (GUSTAFSSON; BETTS, 2010) 174
Table 9 – Obtained results . . . 178
Table 10 – Obtained results . . . 179
Table 11 – Comparing *predicatedLoopUnrolling* with *branchedLoopUnrolling* . . . 180
Table 12 – Used benchmarks from (GUSTAFSSON; BETTS, 2010). 197
Table 13 – Definition of the maximum possible WCET improvement considering the *perfect predictor*. . . . 198
Table 14 – Results of optimizing the subset of benchmarks using five different approaches. . . . 203
Table 15 – Optimality of the five approaches (100% is the optimum).205
Table 16 – List of operation operands (STARKE, 2016) . . . 241
Table 17 – List of supported operations (STARKE, 2016) . . . 246

LIST OF SYMBOLS

bb_i	Basic block i
C	Computation time of a task
D	Relative deadline of a task
di_j	Transition edge from basic block i to j
δ	Pipeline compensation factor
$elbi$	Bound of the outer loop of basic block i
ilb_i	Bound of the loop where header is basic block i
lb_i	Loop bound of basic block i
T	Period or activation interval of a task
t_i	Execution time of a basic block i
x_i	The number of times a basic block i is executed

LIST OF ABBREVIATIONS AND ACRONYMS

ACET	Average-Case Execution Time
BB	Basic Block
BCET	Best-Case Execution Time
CFG	Control-Flow Graph
DAG	Directed Acyclic Graph
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
ILP	Instruction Level Parallelism
IPET	Implicit Path Enumeration Technique
ISA	Instruction Set Architecture
LLVM	Low Level Virtual Machine compiler infrastructure
RAW	Read-after-write
RG	Regular Grammar
RI	Intermediate Representation
SSA	Static Single Assignment Form
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word
WCET	Worst-Case Execution Time
WAR	Write-after-read
WAW	Write-after-write

CONTENTS

1 INTRODUCTION 33
1.1 Basic concepts and motivation 35
1.2 Current scenario 38
1.3 Objective of this Thesis 40
1.4 Original contributions 42
1.5 Organization of the text 43

2 COMPILATION 45
2.1 General aspects 45
2.2 Dataflow analysis 49
2.3 Code generation 52
2.3.1 Instruction selection 52
2.3.2 Register allocation 53
2.3.3 Instruction scheduling 53
2.4 Chapter summary 64

3 WCET ANALYSIS 65
3.1 Basic Concepts 66
3.2 Main existing approaches 70
3.3 Static approach for WCET estimation 72
3.3.1 Value analysis 72
3.3.2 Processor behavior analysis 72
3.3.3 Worst-case calculation or path search 75
3.4 Chapter summary 77

4 OPTIMIZATIONS FOR WCET REDUCTION . . 79
4.1 Dealing with WCEP switches 81
4.2 Characteristics exploited by otimizations 84
4.3 Otimizations to reduce WCET 84
4.3.1 At source code level 85
4.3.2 At assembly level 91
4.3.3 Through code layout 99

4.3.4 Through *scratchpad* allocation 105
4.4 Chapter summary 107

5 EXPERIMENTATION INFRASTRUCTURE . . . 109
5.1 Architecture and reference processor 109
5.1.1 Registers 110
5.1.2 Instructions 110
5.1.2.1 Branch prediction instructions 111
5.1.2.2 Predicated execution of instructions 111
5.1.3 Processor organization 113
5.1.4 Instruction coding 114
5.1.5 Procedure calling conventions 115
5.2 Code generation for the reference architecture . . . 116
5.2.1 Back-end support 117
5.2.2 Code Linking 126
5.3 WCET Analysis 128
5.3.1 Instruction cache analysis 131
5.3.1.1 Reachable and effective abstract state 132
5.3.1.2 Cache accesses classification 135
5.3.2 Pipeline Analysis 136
5.3.3 Worst-case path search 139
5.3.3.1 ILP Constraints 142
5.4 Enabling WCET Reduction Schemes 150
5.4.1 Approach 1: Back-end adaptation To Use WCET Information 151
5.4.2 Approach 2: Code Optimizations Guided by an External Planning Tool 152
5.5 Chapter Summary 153

6 CONTRIBUTION 1: COMBINING LOOP UNROLLING STRATEGIES AND CODE PREDICATION 155
6.1 Introduction 155
6.2 Summary of Related Work 157
6.3 Motivation 158

6.4 Our loop unrolling approach 162
6.4.1 Example 165
6.4.2 Combining Loop Unrolling techniques 166
6.4.3 Ensuring WCET reduction by unrolling factor selection 168
6.5 Evaluation 172
6.5.1 Implementation aspects 172
6.6 Results 173
6.7 Conclusion 176

7 CONTRIBUTION 2: ON THE USE OF STATIC BRANCH PREDICTION 181
7.1 Introduction 181
7.2 The perfect branch predictor approach 184
7.3 A new technique to reduce WCET using branch prediction 186
7.3.1 Considerations on the classic approach 186
7.3.2 The proposed technique 192
7.4 Evaluation of techniques against the perfect predictor 195
7.4.1 Results 196
7.5 Conclusion 201

8 CONTRIBUTION 3: STATIC GAIN POINT IDENTIFICATION AND GAIN TIME ESTIMATION . 207
8.1 Introduction 207
8.2 Related work 208
8.3 Identification of gain time 209
8.4 Case study: applying on an example from the Mälardalen benchmarks 214
8.4.1 Results 215
8.5 Conclusion 215

9 FINAL REMARKS 219
9.1 Publications 222

9.2 Suggestions of future work 224

Bibliography 227

ANNEX 239

ANNEX A – INSTRUCTIONS OF THE TARGET ARCHITECTURE 241

1 INTRODUCTION

A wide range of systems are distinct from the general purpose computing systems due to the need of satisfying rigorous timing requirements, often under the constraint of available resources (AXER et al., 2014), they are generally called real-time systems. The development of a predictable system is concerned with the challenges of building systems whose time requirements can be guaranteed *a priori*. Although, these challenges become even greater when using processors' architectural features for performance increase, as caches and pipelines, which introduce a high degree of uncertainty, making it difficult to provide any kind of guarantee. Parallel to this, there are the tools needed to develop and execute an application, such as languages, compilers, runtime support, communication systems and scheduling, which may further difficult guarantees assertion.

In these systems, the results of computations must be correct not only from a logical point of view, but also must be generated at the right time. In these systems, faults of temporal nature can be considered critical with respect to their consequences. In the literature, real-time systems are classified according to the criticality of their time requirements (FARINES et al., 2000). In *hard real-time* systems, the failure to meet a time requirement can result in catastrophic consequences both in the economic sense as in human lives. When timing constraints are not critical (*soft real-time*) they only describe the desired behavior. The non-compliance with such requirements reduces the utility of the application but does not eliminate it completely nor results in catastrophic consequences.

Real-time systems are present in countless applications, such as in industrial plants, aviation, automotive electronics, telecommunications and space systems. In many of these applications, the complexity of software imposes serious restrictions on the hardware that can be used. This hardware should have sufficient capacity to support the application in question, in addition to be subjected to non-functional design constraints, such as cost and energy efficiency.

Modern general-purpose architectures require, as a basic premise, that programs should run as fast as possible in most times. That is, on average, the execution time of a program should be as small as possible. This average time is usually called ACET - *Average-Case Execution Time*. Although, in some runs, the execution time of the application may be greater when compared to its average case, it will be amortized by the many executions of the program. Average-case prioritization presents certain problems regarding its use in real-time systems. Such systems may require run-time guarantees that are difficult to be obtained or even unfeasible. These guarantees require knowledge of the worst execution time of a program or task on a given processor, which is generally called WCET - *Worst-Case Execution Time*. In a general purpose architecture aimed at the average case, the execution time of a program or task can be so great in the worst case that invalidates the design constraints, or even be impossible to be calculated or estimated with a reasonable effort.

The difficulty of obtaining the WCET is presented by (WILHELM et al., 2008) and is illustrated in Figure 1. According to the figure, we can see how far the WCET can be from the best possible execution time, or BCET - *Best-Case Execution Time*, for a given program. In fact, the WCET calculation problem is undecidable and involves the *halting* problem, then we can only estimate upper limits for its value in practice. Some factors that contribute to the difficulty of obtaining WCET in modern general purpose architectures are pipelines, caches, speculative execution, dynamic branch predictors among other performance enhancement mechanisms.

Currently, there are academic lines that suggest the use of processor architectures oriented for real-time applications (SCHOEBERL, 2009), (SCHOEBERL et al., 2011), (EDWARDS; LEE, 2007). Such architectures adopt hardware features that make analysis related to obtaining WCET estimations simpler and faster.

An important characteristic is that the performance of specific architectures, as the ones aimed at real-time systems, may be closely related to the compiler and compilation techniques employed. Given

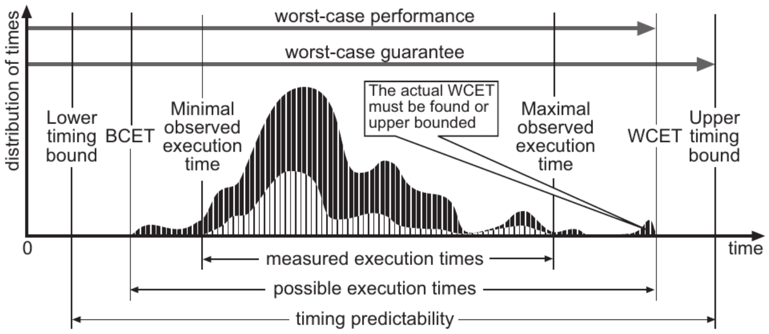


Figure 1 – Example distribution of execution times of a hypothetical program in a general purpose architecture (WILHELM et al., 2008)

the possibility of obtaining the WCET of programs for a specific architecture, we can use this information in an incremental process of optimization, as proposed by (FALK et al., 2006), (FALK; LOKUCIEJEWSKI, 2010) and (HUANG et al., 2012), which are applied to complex architectures, with equally complex WCET analysis. These optimizations are intended to reduce the WCET, since traditional code transformations made by compilers may even increase the WCET of a program (LOKUCIEJEWSKI; MARWEDEL, 2011).

1.1 BASIC CONCEPTS AND MOTIVATION

In control and industrial monitoring applications, automotive and avionics, each function performed by the system is associated with one or a set of tasks (LIU; LAYLAND, 1973). Some of these tasks run in response to external events, while others are activated by system timers or other tasks. Tasks are classified as aperiodic or sporadic. Aperiodic tasks are associated with events which cannot be predicted temporally (when the event may or may not occur). When we know that the event has a minimum time interval between occurrences, we associate this event with a sporadic task. When a task is enabled, it needs to execute a sequence of instructions or perform its computation (associated with the function executed by the task), whose running time in the worst case (***worst-case execution time***) is denoted by C. When there is a limit or time restriction for the task to execute its computation, this is denoted by D, which is the relative ***deadline*** of the task.

In general, tasks activated by timers are called periodic. The activation interval of a periodic task is called period and is denoted by T. When the ***deadline*** of a task is equal to its period, it is said that this is an ***implicit*** deadline. If it is greater than the period then we call it ***arbitrary*** and when it is lower than the period it is called ***restrict***. The most common model used in both literature and in real systems is the periodic tasks with implicit deadlines.

A set of tasks, time constraints and resources, which can be processors, other hardware devices and in-memory data structures (mutually exclusive data) comprise a scheduling problem. Task scheduling can be summarized as the definition of orders in which the tasks can access the resources in respect of all timing constraints specified. The task scheduling is an NP-complete problem.

To schedule a set of tasks, one can construct a fixed and cyclical timeline of execution, which defines the instants in which each task should run. The feasibility of the timeline construction by itself already guarantees the schedulability, but this procedure, which is called ***Cyclic Executive***, is restricted to small sets of tasks. A more general way is to define a scheduling algorithm and an associated schedulability test. If the set is accepted by the test, then it may be scheduled by the algorithm without deadline misses, otherwise the system is not schedulable or nothing can be proved on the corresponding task set.

The parameters typically used by schedulability tests are: activation frequency or period of a task i, denoted by T_i, the above mentioned computation time C_i, and the deadline D_i. A task set is said to be ***feasible*** with respect to a particular type of system, if there is an algorithm that schedule all ***jobs*** (instances of a given task) of all tasks (in all possible sequences) with all temporal constraints being guaranteed. A scheduling algorithm is *optimal* with respect to a type of system if it is able to schedule any set of tasks that can be scheduled by any other algorithm, or otherwise, any task set that is *feasible*.

There are two general categories of schedulability tests. A test is said to be *sufficient* if all task sets that pass the test are guaranteed to be schedulable, but nothing can be said if a set does not pass the test. The second category of test is called necessary, which is simple but not too restrictive. A *necessary* test ensures that, if the task set fails the test, it is certainly not schedulable, but if approved, nothing can be said. This type of test is useful to discard non-schedulable task sets in a practical way.

Contextualizing a more general overview of guarantee providing, (THIELE; WILHELM, 2004) points out that the problem can be related to all layers of a system:

Hardware architecture: This layer holds all aspects related to what is below the instruction set, as microarchitecture and caches. In this layer, predictability relates to variability in the execution time of the program instructions.

Software development for a task: This layer represents all that is necessary for the development of the software which implements the functions of a task. This can involve code synthesis for model-driven development, compilation and all analysis and optimization tools. Nondeterminism is related to the structure and characteristics of the generated code, such as limitation of loops and indirect function calls.

Task level: If the application (tasks of the previous layer) is partitioned as tasks and threads in an operating system, there will be nondeterminism regarding scheduling, memory management and management of various resources. This layer must perform scheduling, as mentioned above.

Distributed operation: It is common for applications to use distributed resources. This layer holds the distributed scheduling and net-work communication. What matters for this layer are the end-to-end deadlines.

As showed above, the WCET parameter is fundamental for the schedulability of a task set. This parameter is strongly influenced by the first two layers, in which we can highlight: processor architecture, the tasks that make up the system and the compiler used in the design. The estimation of this parameter is usually done directly on the executable code of the task and is performed by an analyzer which takes into account the behavior of the processor (cache, memories, pipelines and other factors).

The motivation of this work is the fact that the compiler has information needed to reduce WCET. A compiler can also provide relevant data that are necessary to the proper WCET calculation, avoiding duplicated steps in both tools. Traditionally, compilation and WCET analysis are done isolated in the design and development flow of a real-time system.

However, the compiler builds inherently a large amount of data that would be useful to speed up and simplify the WCET analysis. WCET analysis is also able to generate information that could be useful in the compilation process. All this can be combined with an architecture whose purpose is to conciliate performance with determinism.

WCET reduction is useful for schedulability of the system as a whole. Another utility of this work is cost reduction. With the reduction of tasks' WCET, we can use fewer hardware resources, or increase the number of tasks executing on the same processor.

1.2 CURRENT SCENARIO

Nowadays, the common practice in the real-time system industry is to develop applications according to their usual methodologies, and only at the end of the process, verify whether the timing constraints are met. This constraint verification involves the estimation of the application WCET, which can be done with special tools that are developed for this purpose. In general, WCET calculation tools require applications compiled without any CPI (cycles per instruction) reduction optimization, for two main reasons:

1. Optimization can often increase the WCET (LOKUCIEJEWSKI; MARWEDEL, 2011). We can not predict the effects of unrestricted optimizations in terms of WCET;

2. Optimization can cause the loss of the one-to-one mapping between an object code and its source code. Many analyzers require annotations in the source code to inform constraints that are hard to be extracted from a binary program. With code changes, these restriction/annotations no longer make sense, possibly generating an erroneous WCET. Another factor is that changes can leave the code structure in a difficult way to be understood by the analyzer.

Available tools can employ various approaches to obtain the WCET, as abstract interpretation (aiT[1]), code snippets measurement on real hardware (RapiTime[2]), and even simulation.

A common strategy used in the industry is to explore performance increase characteristics of processors, combined with the imposition of a slack margin whenever the application has a high level of criticality (AXER et al., 2014). The problem with this approach is that timing requirements are possibly not fully guaranteed. Resources can also be wasted, which can be problematic when we have energy constraints, for example.

From an academic point of view, there are projects like PREDATOR[3], MERASA(UNGERER et al., 2010), PRET (EDWARDS; LEE, 2007; LIU et al., 2012) and T-CREST[4] which aims at the adequacy or the building of architectures for determinism and timing control.

In relation to advances in compilation, there are some works that focus on reducing the WCET by applying optimizations in specific con-

[1] http://www.absint.com/ait/
[2] http://www.rapitasystems.com/products/rapitime
[3] http://www.predator-project.eu
[4] http://www.t-crest.org

texts. A compiler targeted to WCET reduction is presented in (FALK et al., 2006). This compiler is coupled to the aiT WCET analyzer to discover possible optimization potentials. In (ZHAO et al., 2005) it is also used a compiler coupled with a WCET analyzer for optimization purposes.

Some techniques rely heavily on the target architecture of the compiler. In (ZHAO et al., 2005), an analyzer provides information concerning the worst-case paths so that the compiler can organize the code in order to reduce branch penalties. The architecture used is the StarCore SC100, a DSP with different branch penalties. Another technique is presented in (FALK; KOTTHAUS, 2011), which aims to place basic blocks in memory to optimize the worst-case cache behavior. This strategy was already used to reduce ACET. In this paper, the authors use the concept of formal cache model which can capture the behavior of all type of caches. This work was the first to consider code positioning seeking to reduce WCET through cache behavior.

There are other works that try to adapt classic compiler optimizations to the WCET reduction context. Techniques such as *loop unrolling* (ZHAO et al., 2006) (LOKUCIEJEWSKI; MARWEDEL, 2009) and instruction scheduling approaches (LOKUCIEJEWSKI et al., 2010) (LOKUCIEJEWSKI; MARWEDEL, 2011). Other optimizations will be covered in the related work Chapter of this Thesis.

1.3 OBJECTIVE OF THIS THESIS

The objective of this work is to contribute to aspects related to the compilation for real-time systems, whose primary goal is the WCET reduction or improvement of aspects related to schedulability. The thesis to be demonstrated is that the close coupling of a compiler with a WCET analyzer can benefit both the analysis and synthesis of an executable program or a complete system to a deterministic architecture. The use of a deterministic architecture represents an important feature of this work. The development of WCET analyzer infrastructure for the target architecture is also part of this work.

Among the elements related to the compiler that are essential to reduce WCET, we can mention:

- Mechanisms for WCET calculation of programs in process of compilation. This implies the coupling of the compiler with the analyzer developed.
- Identification of potential points to be benefited by optimizations. This involves interpretation of the analyzer results.
- Discarding code changes that increase the WCET. Again, decisions should be made based on successive analyzes.

In addition to the related elements, we can highlight the process efficiency. Using an architecture designed for real-time applications allows the use of a much faster and accurate analyzer, which aims to bring efficiency to the process.

The architecture that was explored in this thesis, which was developed in another work from the same research group (STARKE, 2016), imposes additional challenges. It uses techniques such as VLIW - *Very Long Instruction Word* to increase performance. This technique requires the static exploration of parallelism between instructions by the compiler, unlike what usually happens in general-purpose processors. For determinism, techniques such as execution of predicated branches, predicated execution, scratchpad memory and absence of data cache are present in the architecture. However, instruction cache is used because it is more predictable than its data counterpart, because its access pattern depends on the instruction sequences and can be easily discovered *a priory* by an analysis tool.

Though the used architecture is based on a commercial ISA, there is no free compiler available for this. Then, the implementation of a fully functional code generator was required as a prerequisite for the realization of this work. As the architecture was developed in parallel to the compiler development, we had the exact measurement of what was needed in both sides of the infrastructure, avoiding unnecessary engineering and development.

1.4 ORIGINAL CONTRIBUTIONS

The contributions of this thesis to the state-of-the-art are:

1. The proposition of a different way to perform loop unrolling on data-dependent loops using code predication targeting WCET reduction, because existing techniques only consider loops with fixed execution counts. We also combine our technique with existing unrolling approaches. Results showed that this combination can produce aggressive WCET reductions when compared with the original code.

2. Regarding static branch prediction techniques, we show that a very small or even no gain can be obtained with new optimization techniques targeted to worst-case execution time reduction. To achieve this objective, we compare several techniques against the perfect branch predictor. This predictor permits to estimate the maximum WCET reduction that can be obtained with static approaches. In addition to the classic technique of the literature, we include in the comparison a new WCET-centered technique which acts as a brute force approach to bring the results as close as possible to the perfect predictor. The comparison also includes standard compiler techniques not directly oriented to WCET reduction. As result, we show that the techniques considered in this thesis are close to the optimal result obtained by the perfect predictor. We also show that our technique produces slightly better results than the other techniques. As a secondary contribution, we show that WCET-unaware techniques can also be used in real-time environments because they present good results and low complexity. We evaluate prediction techniques using a set of examples from the Mälardalen WCET benchmarks.

3. One problem of the WCET is that it is relative to a single execution path, specifically the worst-case execution path (WCEP). When a real-time application executes over a path different from

the WCEP, its execution time will be probably smaller than the WCET. The difference between the WCET of a task and its actual execution time is called gain time. We propose a technique that finds specific points of a program (called gain points), where there will be an amount of statically estimated gain time in case that path is taken by the execution. As a case study, we present the gain time obtained by applying our strategy to one benchmark from the Mälardalen WCET benchmarks suite. For the selected benchmark, several gain points were identified and some of them with a significant amount of statically detected gain time.

1.5 ORGANIZATION OF THE TEXT

This work is organized as follows: Compilation aspects are presented in Chapter 2. Approaches to WCET analysis are covered in Chapter 3. Program optimization considering WCET reduction is the theme of Chapter 4. The experimental infrastructure is presented in Chapter 5. Our original contributions are presented in Chapters 9.2, 7 and 8. Finally, Chapter 9 presents our conclusions and final remarks.

2 COMPILATION

Compilers are tools present in current software development processes. They are responsible for converting the representation of a program in the humanly understandable way to useful representations for computers or virtual machines.

In the current flow of software development, compilers have prominent space because they are able to automatically apply optimizations to improve the quality of executable code. But even modern compilers are not able to generically optimize code for real-time systems because they have no way of quantifying the impact of such optimizations over time aspects. The WCET-oriented compilation is a recent approach, with few published works. In the next sections, we will cover basic compilation issues, leaving real-time aspects to the next chapters.

The major emphasis of this chapter lies in data flow analysis, which is a technique related to compilation, but can be extrapolated to other areas that depend on this type of information, such as WCET analysis. We will also cover instruction-scheduling techniques for VLIW architectures (*very long instruction word*). This emphasis was given due to the use of such an architecture, as presented in the previous chapter.

2.1 GENERAL ASPECTS

According to (AHO et al., 2008), compilation is a mapping between a program written in a high level language and a program written in a semantically equivalent machine language. This mapping process can be seen as two steps, which are:

Analysis: This step breaks the program into parts, in which grammatical structures are imposed. The next step is to convert these structures into an intermediate representation of the program. In this step, syntactic and semantic analyzes of the program are performed. The symbol table construction is also part of this step. A symbol table stores information about the symbols (variables, functions, etc.) and their respective validity scopes (global, local, etc.).

Synthesis: Starting from the intermediate representation and the symbol table, the synthesis step generates the final representation of the program, which can be executed on machines (real or virtual) that implement some specific instruction set architecture (ISA).

Traditionally, all steps of the compilation process are organized in passes. In this way, each pass perform transformations from one representation to another. Passes can perform transformations in the context of a specific intermediate representation, such as those that perform optimizations for example. From this perspective, a traditional compiler can be viewed generically as having the structure shown in Figure 2. In this figure the main passes and the information flowing between them are highlighted. Those informations are called program representations, and can be:

Source code: comprehensible representation for humans, that is, code written in some programming language;

Syntax tree: a tree (or graph) that represents the syntactic structure, which is related to the chosen language. In this representation, operands are categorized as child nodes of nodes representing operations;

Intermediate Representation (RI): the internal compiler code representation is generally independent of any specific language and/or architecture. The intermediate representation may be similar to an *assembly*. Such assembly may have several properties, such as being in the SSA form - static single assignment form (CYTRON et al., 1991). The SSA property defines that we can assign a value to a variable only once. When we have branch and join nodes, we add a special form of assignment called $\phi - function$, to unify the two defined (in each path) variables/register in one.

Target code: represents the generated code for a given target architecture.

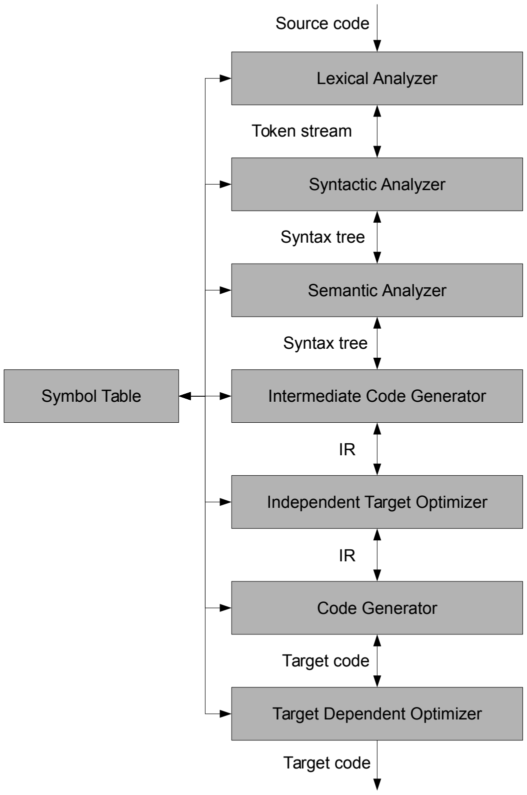


Figure 2 – Common pass structure of a compiler

According to the grouping of passes previously presented, we can describe the function of each specific pass:

Lexical Analyzer: also known as lexical analysis or *scanning*, has the responsibility of reading the source code and grouping the characters into sequences that make some sense, which are called lexemes. For each lexeme produced, the lexical analyzer produces an output token, which is passed on to the next phase, which is the syntactic analysis. This analyzer usually uses regular expressions (or regular grammars - RGs) implemented in the form of deterministic and finite automatas;

Syntactic Analyzer: the parser processes the *tokens* obtained by the lexical analyzer into an intermediate representation, which is usually a syntax tree. This transformation uses context free grammars and its recognizers are implemented in the form of stack automata for identification of language patterns. If some pattern can not be recognized, the tree can not be terminated and a syntactic error is then reported. The syntactic analysis also populates the symbol table of the program;

Semantic Analyzer: the semantic analyzer operates on the syntax tree in conjunction with the symbol table. This analyzer checks the program for inconsistencies, such as type mismatch, parameter checking in procedure calls, and so on. Additional information can be annotated in the symbol table and in the syntax tree;

Intermediate Code Generator: syntax trees are suitable for syntactic and semantic analysis. More internal compiler operations generally require lower-level intermediate representations. Such operations often resemble the assembly of a virtual architecture. There are numerous representations, such as three-address code, where each operation has exactly three operands per instruction;

Target Independent Optimizer: this component is responsible for improving the quality of the code that will be generated later. Optimizations can have several goals: performance, code size reduction and etc. The range of optimizations applied by a compiler can vary from version to version of the tool or by user demand (optimization level);

Code Generator: the code generator transforms the optimized intermediate representation into a version that can be executable on the target processor. The assignment of registers to variables, which is called register allocation, is also part of this process;

Target Dependent Optimizer: this component, if it exists in the compiler, can perform optimization that are specific to the target architecture. Optimizations involving code scheduling for *Instruction Level Parallelism* improvement can also be attributed to this optimizer. Punctual changes of instruction sequences (*peephole* optimizations) can also be applied by this component.

An alternative classification of compiler components considers architecture and language dependency. In this classification, the part called *front-end* represents the portion of the compiler that is language dependent, and must be ported when a new language is developed. At the other side of the compiler is the *back-end*, which has the information related to the target architecture, and is responsible for generating code. This part of the compiler must be ported to all supported architectures. The most internal part is called *middle-end*, which concentrates the compiler part that is independent of both language and architecture. Generally, this part is responsible for making generic transformations/optimizations and being the point of contact between the adjacent parts.

2.2 DATAFLOW ANALYSIS

Several parts of a compiler depend on what is called data flow analysis. This type of analysis is also useful for WCET calculation techniques, which is the subject of the next chapter. Such analyzes extract relevant information about the flow of data along the paths of execution of a program. In a basic block, the control go from the beginning to the end, without interruptions and flow deviations, more formally (MUCHNICK, 1997):

Definition 1. *A basic block (BB) is a sequence of consecutive machine instructions in which the flow of control enter at the beginning and leaves at the end, without jump targets between them.*

Starting from the definition of basic block, we can formally define the structure that represents the possible paths that a program can cross. Such a structure is called the Flow Control Graph (CFG). Given the definition of the component that represents the node of the CFG, we can then formally define it:

Definition 2. *A control flow graph $CFG = (V,E)$ is a directed graph, where V is a set of nodes, representing basic blocks, and $E \subseteq V \times V$ is the set of edges, representing the flow of control.*

The edges representing the flow of control model the transitions (calls, branches, jumps and sequential execution) between pairs of basic blocks.

The data flows can be declared in terms of values entering and leaving the blocks. For a given block B, the input and output data flow values can be defined by $IN[B]$ and $OUT[B]$, respectively. $IN[B]$ and $OUT[B]$ can be calculated by considering $IN[s]$ and $OUT[s]$, for all instructions s of block B . The relationship between the data flow values before and after the execution of instructions or basic blocks are represented by *transfer functions*.

Transfer functions operate in two directions. Forward functions convert data flow values that enter an instruction to the values that leave it:

$$OUT[s] = F_s(IN[s])$$

On the other hand, backward functions convert data-flow values after the instruction to values before the instruction:

$$IN[s] = F_s(OUT[s])$$

The relationship between statements of a basic block follows the following constraints, assuming that such a block is composed of the statements $s_1, s_2, ..., s_n$:

$$IN[s_{i+1}] = OUT[s_i], \forall i = 1, 2, ..., n-1$$

For basic blocks, $IN[B]$ corresponds to $IN[s_1]$, that is, the input of the first instruction of the basic block. The output $OUT[B]$ corresponds to the output of the last instruction $OUT[s_n]$. Since f_{si} is the transfer function of the i instruction of the basic block B, then the transfer function of such a block can be written as $f_B = f_{sn} \circ ... \circ f_{s1}$.

The relationships between the begin and end of basic blocks must take into account the possible existence of several successor and predecessor basic blocks, as well as the direction of the analysis. These relationships can be:

Forward Flow Relationship: When a forward flow problem is being solved, the relationship between the begin and end of basic blocks is:

$$OUT[B] = f_B(IN[B])$$
$$IN[B] = \bigcup_{P \quad predecessor \quad of \quad B} OUT[P]$$

Analysis of reachability of definitions and available expressions are examples of forward flow problems.

Backward Flow Relationship: On the other hand, when we are solving a backward flow problem, the relationship is given by:

$$IN[B] = f_B(OUT[B])$$
$$OUT[B] = \bigcup_{S \quad predecessor \quad de \quad B} IN[S]$$

Live variable analysis is a backward flow problem.

Generally, iterative algorithms are used to solve data flow problems. Some important information that can be obtained by data flow analysis are:

Reachability definitions: these definitions basically tell us if any definition for a variable x reaches a point p in a program. According to (AHO et al., 2008), a definition d (dependent on x) reaches a point p if there is any path starting from d to p, such that d is not *dead* along that path. If there is any other definition of x along the path, the definition d becomes dead.

Available expressions: the available expression analysis tells us whether a given expression, such as $x+y$ for example, is available at a point p in a program. The expression $x+y$ will effectively be available in p if it has ever been calculated before p and the values of x and y have not been redefined.

Live variable analysis: The live variable analysis says that, if for a variable x and a point p, such variable could be used in some path starting at p. If a variable is used in a path that starts at some point p, it is said to be *alive* at that point, otherwise it is considered *dead*. This is a backward flow problem, and has numerous utilities, such as guiding the allocation of registers.

2.3 CODE GENERATION

This section addresses aspects related to code generation, such as instruction selection, instruction scheduling for VLIW architectures and register allocation.

2.3.1 Instruction selection

The instruction selection aims to convert the intermediate representation into valid instructions of the architecture. This step precedes the instruction scheduling register allocation phases. The selection can be done in several ways, the most common strategy consists on the use of patterns. Such patterns convert a piece of the intermediate representation (usually represented in tree) into a sequence of machine instructions.

2.3.2 Register allocation

Register allocation (AHO et al., 2008) consists of assigning a small set of registers to the variables of a program (possibly large set). It can be done in several granularities, such as locally in basic blocks, whole functions, or considering function boundaries. The problem of register allocation is isomorphic to the problem of graph coloring. According to this isomorphism, variables are represented by nodes in a graph, and the edges represent interferences between them. There is interference between two variables when both are considered alive simultaneously.

With a constructed graph, then, we have a problem of K-coloring, where K represents the number of available registers. When we can not colorize the graph with K colors, we must remove edges by choosing variables to be put into memory (*spilling*). With a variable in memory, the removal of certain edges is possible because the liveliness interval of that variable is reduced. Although graph coloring is an NP-complete problem, there are good heuristics in the literature for the case of register allocation.

2.3.3 Instruction scheduling

Instruction scheduling is the establishment of instruction ordering and/or grouping of instructions with the purpose of improving the execution of those instructions in a particular processor. In superscalar architectures, the discovery of instruction-level parallelism, usually called ILP, is done directly in the processor through out-of-order execution techniques, for example. However, this process can be facilitated by the compiler's action, which can sort/schedule instructions so as to expose such parallelism more explicitly. On the other hand, VLIW architectures do not have automatic ILP extraction mechanisms for instruction flows, and this task is entirely attributed to the compiler developed for the architecture. There are basically two families of instruction scheduling techniques:

Local: In these techniques, ILP is discovered at the basic block level. In this type of scheduling, the code can be rearranged or scheduled into sets of operations that do not violate data precedence and do not use more processor cycles than available. Scheduling instructions in basic blocks is an NP-complete problem, so we can expect that the optimal solution or scaling will be exponential to the number of instructions in a basic block.

The most widespread local technique is *List Scheduling*, which is a heuristic process that generates good results. In this technique, we assign priorities the operations of a basic block. At each cycle, operations with higher priorities are scheduled, with their predecessor operations already scheduled and their processor resource constraints met. This technique is based on the use of a *dependency graph* (LOKUCIEJEWSKI; MARWEDEL, 2011) for each basic block:

Definition 3. *(Dependency graph) A dependency graph $D = (V, E)$ of a basic block B is a directed acyclic graph (DAG), where nodes represent instructions $i \in B$, and edges $E \subseteq V \times V$ connect two nodes v_i and $v_j \in V$ if, and only if, there is a data dependency between v_j and v_i. The dependencies can be:*

True dependency or read-after-write (RAW): *v_i writes an operand that is read by v_j.*

Anti-dependency or write-after-read (WAR): *v_j writes an operand that is read by v_i.*

Output dependency or write-after-write (WAW): *v_i and v_j write to the same operand.*

If the last statement of the basic block $i_{last} \in B$ is a branch or call instruction, edges between all previous statements and i_{last} are added to maintain control flow. Otherwise, the scheduling algorithm could move that instruction to any place in the basic block, violating its definition.

There are several approaches to prioritizing instructions. A simple but good heuristic is *highest levels first*. This heuristic assigns priorities to the operations according to the longest chain in the data dependency DAG, starting at the specified operation and ending at a leaf node.

As an example of the application of list scheduling, we can consider the example given in Table 1, which has been adapted from (FISHER, 1981). This table presents a set of 5 basic blocks, each containing a sequence of instructions. The demand for resources associated with each instruction is also presented in the table. For this example, we assume that the control flow graph of Figure 3 represents the structure of the program.

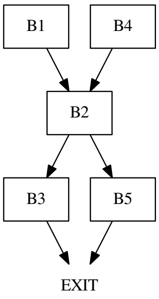


Figure 3 – Control flow graph of the example

Figure 4 presents the DAG of precedences for each basic block of the example. By assigning priorities to *highest levels first*, we get that the instructions above in the DAG will have higher priority for execution in the scheduling process.

Considering the priorities and execution precedence of instructions in Figure 4, together with the resource demands enumerated in Table 1, we can obtain the scheduling shown in Table 2. This table represents, for each basic block, the sequence of instruc-

Table 1 – Instructions per basic block and resource demand for the example

	R1	R2	R3	R4
Block B1				
I1		X	X	
I2	X			
I3	X			
I4		X	X	
I5	X			
Block B2				
I6	X		X	
I7		X	X	
I8			X	
Block B3				
I9				X
I10				
I11				X
I12				X
I13				X
I14				X
Block B4				
I15				X
I16				X
Block B5				
I17	X			

tions that are dispatched to execute each cycle, considering an architecture that can execute two instructions simultaneously. In the example shown, two instructions can execute simultaneously only if they are independent in both data usage and processor resources.

Local techniques tend to not be very efficient because, in general, there is not much parallelism available if isolated basic blocks are considered. To overcome this deficiency, global techniques were proposed.

Global: Global techniques schedule different basic block instructions simultaneously. These techniques operate by selecting a large set

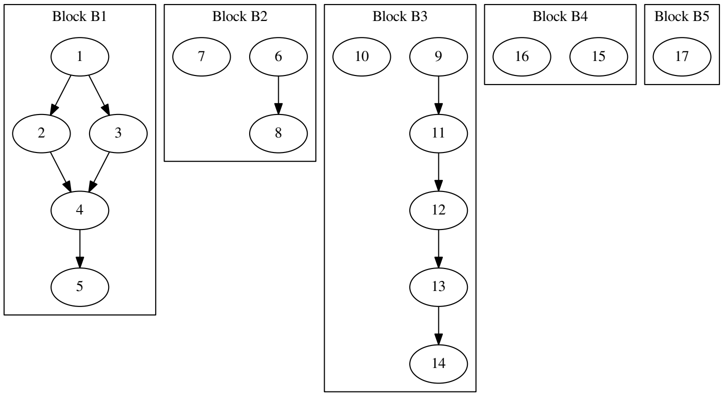


Figure 4 – Precedence DAG for the example

of instructions, which is typically larger than an isolated basic block. This set is then scheduled with local techniques, such as *list scheduling*. Global approaches can vary in terms of how this set of instructions is constructed and how consistency is maintained in relation to divergent flows. Consistencies must be considered when moving instructions to above or below branches or branch targets, since such operations can change the semantics of the program. When scheduling decisions are global, some schedulings can shorten the execution time of a path, while others paths can have their execution time increased.

Considering the global scheduling of instructions, we will briefly describe the main techniques available in the literature for exposing ILP to processors.

Primitive methods (menu): According to (FISHER, 1981), scheduling for VLIW has origins in microcode compression techniques (vertical code transformation in horizontal), where programmers move operations from one basic block to another to generate sets op-

Table 2 – Scheduling obtained for the example

Cycle	Instruction
Block B1	
1	I1
2	I2
3	I3
4	I4
5	I5
Block B2	
1	I6
2	I7
3	I8
Block B3	
1	I9,I10
2	I11
3	I12
4	I13
5	I14
Block B4	
1	I15
2	I16
Block B4	
1	I17

erations that could be executed simultaneously. This process, usually called the menu method, was guided by a set of rules applied to flow graphs, based on dependencies between instructions and register liveliness. Although the idea was to do manual scheduling of code, the first automations of this process soon appeared ((DASGUPTA, 1979) and (PATTERSON et al., 1979)). The following constrains must be considered when we employ the menu method:

1. Only code without loops is considered;
2. Each basic block is compacted separately;
3. Basic block execution orders are formed. This can be simply block lists with the property that, if one block is executed

during one program execution, the other will also be;

4. Blocks are examined in the order formed in the previous step, and legal moves from the current block to blocks previously examined are considered. Movements are committed if they save processor cycles.

The rules for moving instructions across basic block boundaries are:

- If the instruction is moved down from a lateral entry (branch whose entry is in the execution *trace*), then all instructions between it and the old entry point should be copied to the lateral entry. An example of this type of movement is shown in Figure 5.

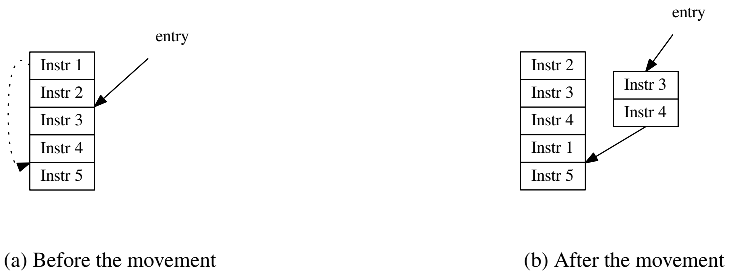
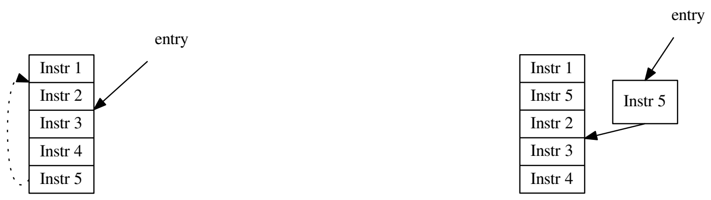


Figure 5 – Example showing the movement of instructions to above lateral entry

- If the instruction is moved to above a lateral entry, it should also be copied to the lateral entry. An example of this type of movement is shown in Figure 6.

- If an instruction *i* is moved to below from a lateral exit and the flow values defined by it are alive for this lateral exit, then *i* must be copied somewhere between the exit and future uses of the values;



(a) Before the movement (b) After the movement

Figure 6 – Example showing the movement of instructions to below lateral entry

- Instruction moves to above lateral exits will only be allowed if the data flow values are not alive for this exit. This restriction can be alleviated if speculative execution is available.

Trace scheduling: The trace scheduling technique operates on *traces* or execution paths in programs, rather than basic blocks. A *trace* is a sequence of loop-free instructions that can be executed continuously for a certain choice of data. A trace can be formally defined:

Definition 4. *(Trace) There is a $followers$ function such that for a given m instruction, the set $followers(m)$ gathers all instructions that can be executed after m. If m_i is in $followers(m_j)$, then we say that m_j is a leader of m_i. If there is more than one instruction in $followers(m)$, then m is a conditional branch. Then, a trace can be defined a sequence of instructions $(m_1, m_2, ..., m_t)$ such that for every j, $1 \leq j \leq t-1, m_{j+1}$ is in $followers(m_j)$.*

The scheduling algorithm consists of successively selecting uncompressed traces that are most likely to execute. After compression, the rules of the menu method force the duplication of operations/instructions to locations outside of the current *trace*. For each *trace*, the algorithm is summarized in 3 steps:

1. Select the *trace* as the most frequently executed path;
2. Schedule the *trace*. This phase involves constructing the dependency DAG, assigning priorities to the instructions (using, for example, *highest levels first*) and subsequent scheduling with list scheduling;
3. Bookkeeping phase. This phase performs the verification and possible duplication of instructions in *traces* outside the current one, in order to maintain the original semantics of the program. The rules used are the same as the menu method.

There are several ways to incorporate loops in *trace scheduling*. A simple way is to schedule (or compress in the original terminology) one loop at a time, in the order $L_1, L_2, ..., L_p$. Each time a loop L_i is ready to be scheduled, the loops L_j included in this loop, being $j < i$, will already have been compressed. Other more powerful methods can be applied by considering the movement of instructions out and into the loops.

Superblock scheduling: Trace scheduling schedules instructions by ignoring control flow transitions (exit branches and trace entry), so checking and correcting consistency is necessary to make sure the code that is outside the *trace* executes correctly. This *bookkeeping* has high complexity since all instruction moves need to be checked, and in some cases there may be a need to insert code into other *traces*.

With the idea of reducing the complexity of *bookkeeping*, *superblock scheduling* (HWU et al., 1993) was proposed. Thus, this technique can be seen as an improvement in trace scheduling. A super block is a *trace* that has no lateral entries, or branch targets. Lateral entries make it difficult to apply optimizations, which is the main motivation of the technique.

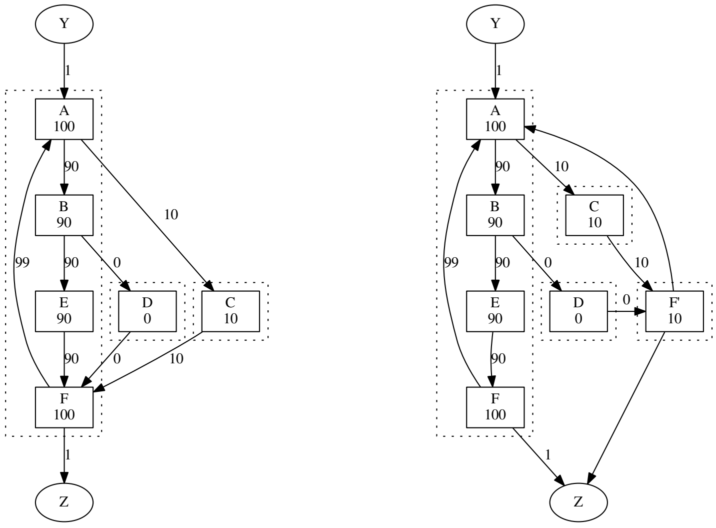
The construction of the super block follows two steps:

1. Identification of *traces*, using static code analysis, or execution *profiles*;
2. Tail duplication. At this stage, any lateral entry to the super block is removed by duplicating the super block's tail, starting from the point of entry to the end. All side entries are directed to the duplicated blocks, these blocks can be added to the end of the function or method. Basic blocks in a super block need not be consecutive in the code as a whole, but the restructuring improves *cache* performance.

The formation of a super block for a program segment containing a loop (HWU et al., 1993) is shown in Figure 7. The control flow graph is valued at nodes and edges. The count of each block represents the execution frequency obtained for the basic block (*profile* or static analysis) and the count of edges represent the frequency of transfer of flow between blocks. Among all the possible paths, the most frequent one is the one that surrounds the blocks {A, B, E, F }, so this path is the ideal candidate for superblock formation and is represented in Figure 7a. From the figure, we can see that there are two side entries in basic block F, which are eliminated by duplicating such a block, as shown in Figure 7b.

After the construction of the super block, some optimizations can be done before proceeding with the scheduling itself:

Superblock enlargement optimizations: The purpose of these optimizations is to increase the size of superblocks often executed in order to expose a greater number of instructions to the scheduler. The greater the number of instructions, the greater the chances of the scheduler to find independent instructions. An important point is that these optimizations only increase superblocks of interest, keeping the



(a) Trace selection (b) Tail duplication

Figure 7 – Superblock formation example

overall code expansion under control. Among these optimizations, we can mention: ***branch target expansion***, ***Loop peeling*** and ***loop unrolling***.

Dependency removal optimizations: The second type of optimization aims at the elimination of dependencies between instructions in frequently executed superblocks, increasing ILP. These optimizations can also induce controlled code expansions. Among these optimizations, we can mention 5: ***Register renaming***, ***operation migration***, ***induction variable expansion***, ***accumulator variable expansion*** and ***operation combining***.

Once the optimizations are done, it follows the scheduling. The scheduling of a superblock consists of two steps: construction of the dependency graph and *list scheduling*. Dependencies on data, control, and branches are represented in this structure. Information about the architecture used can be incorporated into the scheduling process, such as instruction latencies and restrictions related to processor resources.

2.4 CHAPTER SUMMARY

Compilation is a key aspect in computing nowadays. There are compilers for the most diverse languages and architectures. The purpose of this chapter is to provide an overview of the compilation process, with an emphasis on data flow analysis and instruction scheduling. It is not the purpose of this chapter to present an in-depth overview on compilation, only an overview of the phases of the compilation process, with emphasis on techniques related to the doctoral proposal and those necessary for the development of experimental infrastructure.

Data flow analysis consists of techniques that allow we to discover certain properties and information that are useful to compilers. This work proposes to perform code transformations in the context of real-time systems, and most of the transformations require information obtained by data flow analysis. In addition, this type of analysis is useful for WCET analysis tools for example. Cache analysis can be performed with the help of this type of technique.

Another topic addressed was instruction scheduling. Scheduling Instructions is intended to unambiguously expose the order and parallelism of instructions to VLIW processors, or even to help in the case of superscalar processors. All of these concepts are important for the context of this thesis, since the architecture used is dependent on the scheduling of instructions made by the compiler.

3 WCET ANALYSIS

This chapter aims to introduce several WCET analysis techniques. Some techniques mentioned here are used in the experimentation infrastructure, as will be presented in Chapter 5. Beyond introducing WCET techniques, analysis complicators are also presented in this chapter. Analysis complicators are characteristics of both the processors and programs under analysis that make difficult to obtain a WCET. The presence of complicators certainly requires the use of more sophisticated techniques.

The main objective of the WCET analysis is to determine the worst-case execution time of a task or program when executed on a certain hardware. Analyzers can also provide estimates for code segments that execute on interrupt handlers, for example. The worst execution time is determined by the execution time of the instructions of the program that are present in the worst-case execution path (WCEP). Often, such a path is only discovered at the end of overall analysis process. The most difficult part of this type of analysis is certainly the modeling of processor behavior, considering all associated mechanisms.

The problem of obtaining WCET is undecidable and to solve it means to solve the halting problem. So what we do in practice is the calculation of estimates for a restricted set of programs that meet a set of constraints: the program must terminate, then recursion levels and number of loop iterations must be explicitly limited. According to (ENGBLOM; ERMEDAHL, 1999), for a WCET estimate to be valid it must be *secure* (not an underestimation), and to be *useful* it should be as small as possible (small overestimation). Usually estimates take into account that the code executes without interruption and without *background* activities. Such aspects should be addressed at a higher level, such as in scheduling for example.

The usual approach consists in to to split WCET analysis into different subtasks. Some subtasks deal with flow-related characteristics and their control, while others with the execution time of instructions on the chosen hardware, taking into account *cache* and *pipeline*, for example. In the next section we will present concepts related to the WCET analysis. Some of these concepts are common in the compiler literature.

3.1 BASIC CONCEPTS

The main artifacts of the subtasks that make up the WCET analysis are the executable code itself, the Control Flow Graph (CFG) and the Procedure Call Graph, which are often integrated into a single structure. Each node of the CFG represents a single basic block. The definition of Basic Block and Control Flow Graph was given in Section 2.2 from Chapter 2.

Generally, a CFG represents the flow of control of only a single function. However, the control flow of an entire program can be represented by the Interprocedural Control Flow Graph (ICFG), which is a composition of all function calls from the CFG of the main procedure of the program, as described by (WILLIAM; BARBARA, 1991).

This graph contains all possible execution paths of a program, from average execution paths to the path that generates the worst-case execution time. Formally, we can define a path as follows:

Definition 5. *A path in a $CFG = (V, E)$, from a node v_0 to v_k, is a sequence $\{v_0, v_1, ..., v_k\}$ of nodes such that $(v_{i-1}, v_i) \in E$, for $i = 1, 2, ..., k$.*

Considering an execution path, we can define the concept of *dominance*:

Definition 6. *Node n_i dominates n_j, written n_i dom n_j, if every path from the source to n_j includes n_i.*

Another concept widely used in both compilation and WCET techniques is loop. A simple definition of loop is the following (AHO et al., 2008):

Definition 7. *(Loop) A loop is a strongly connected component of the G graph representing the CFG. A loop consists of a single header, which is the entry point of the loop. This header dominates all other nodes in the* *loop. There may be different edges that return from inner nodes of a loop to the header, as well as different loop exit edges. Every return edge is related to at least one loop.*

A natural loop relative to a return edge $n \rightarrow d$ is the set of nodes dominated by d, and that can reach n. Natural loops are more easily understandable by WCET analyzers than unrestricted loop structures.

There are many ways to build the CFG of a program. We can build such a graph from the source code of the program, from the executable code, or in a less usual way, extract this information directly with the compiler that generated the program executable. Some information is usually annotated in CFG's, such as the limits of execution of loops, which are extracted by analysis or manually informed by annotations in the source code.

Considering the control flow graph and the behavior of the processor used, WCET analysis may have several complicators. According to (WILHELM et al., 2008), some complicators are:

Data-dependent flow: The WCET of a task is linked to a particular execution path/flow described in the CFG. If this flow is dependent on input data, and if such data is known, then the problem can be solved easily by measuring the execution of the program in hardware. The problem is that the data that leads to the worst execution flow are generally not known and initial states of hardware to start the measurement are extremely complex for such an approach.

Context-dependent execution times: Old analysis techniques assumed that execution times were context independent. Such an assumption relies from the fact that older processors document latencies of instructions in their manuals. For example, we can consider the basic block C with two predecessors A and B. With context-independent execution times, C will always have the same execution time, regardless of whether it has been reached by A or B. However, with modern processors, this information is no longer available due to a series of complications. The main complicators are certainly the *caches* and *pipelines*. Currently, the behavior of the processor should be analyzed as a specific subtask of the WCET analysis.

Timing anomalies: The high complexity of current processors also drastically affects the applicability of techniques that analyze processor behavior. Modern processors suffer from effects called timing anomalies. Such effects are called anomalies because they are counter intuitive, where a better local case may induce a worse global case. Components of processors that can cause anomalies are branch predictors with speculation, and mechanisms of out-of-order execution.

Figure 8 shows an example of how speculation can cause timing anomalies. In this example, a *cache hit* induces a larger global execution time for the example. This is counterintuitive, but in this example a *cache miss* from A prevents the speculative unit from taking C from *cache*. When a *hit* occurs in the *cache* in A, the speculative unit tries a path that is not what will actually execute, generating a *miss* in C, costing more than the first *hit*.

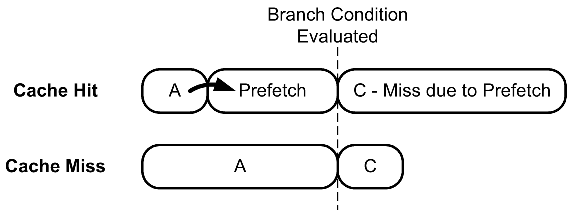


Figure 8 – Example anomaly caused by speculation (WILHELM et al., 2008)

As stated earlier, out-of-order execution may cause anomalies. Such anomalies are called scheduling anomalies. In this type of anomaly, the same sequence of instructions may have different execution times, depending on the availability of resources in the processor, such as pipeline units/memory access. This availability is queried by the dynamic instruction scheduler of the processor when generating the best execution orders for a certain instant. An example of such an anomaly is shown in Figure 9. In this example, the fact that A, which uses ***Resource 1***, takes longer to execute in the first case, induces a shorter global time. In the second case of the example, the fact that A takes less, in conjunction with the data dependencies between instructions and availability of processor resources, generates an instruction scale whose time is longer.

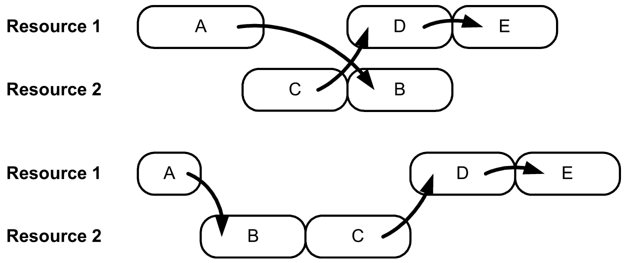


Figure 9 – Example of scheduling anomaly(WILHELM et al., 2008)

Timing anomalies invalidate assumptions that consider worst local cases, since they do not guarantee worst global cases. Another relevant factor is that with time anomalies it becomes not safe to obtain the WCET from measurements with worst input data and worst initial state of the processor, because it is not known which state will culminate in the general worst-case of the processor. Anomalies force the processor analysis to follow several successors in the solutions search space, whenever a non-deterministic processor state is encountered. This can lead to extremely large state space, even for small programs. A significant part of modern processors are considered non analyzable, which can be partly attributed to the existence of timing anomalies.

A special case of timing anomaly is the so-called domino effect

(WILHELM et al., 2009). A processor exhibits domino effect if there are two states s and t, such that the difference in execution time of a program when started in these states is arbitrarily high, not being limited by a constant factor. An intuitive example is a loop, where iterations never converge to the same hardware state, and the difference in execution time increases with each iteration.

We can classify the architectures according to the presence of timing anomalies and domino effect(AXER et al., 2014):

Completely compositional in relation to time: If the abstract model of an architecture does not present timing anomalies, this can be classified as completely compositional with respect to time. So the analysis can securely consider worst-case local paths. An example of this type of architecture is the ARM7, which can be analyzed in a very simple way. When a *hazard* occurs in the *pipeline*, all components suffer *stall* until the resolution of this *hazard*. Then, all analyzes for different components can be done separately, whose results are composed at the end.

Compositional with limited and constant effects: This type of architecture has temporal anomalies, but does not exhibit domino effects. In this case, analyzes should consider all possible paths. Infineon TriCore is considered a processor of this category, although this has never been proven.

Non-compositional: These are architectures that exhibit both temporal anomalies and domino effect. The PowerPC 755 is an example of this type of architecture. For this type of architecture, time analysis must follow all paths, since a local effect can influence future execution arbitrarily.

3.2 MAIN EXISTING APPROACHES

According to (LOKUCIEJEWSKI; MARWEDEL, 2011), techniques for WCET estimation can be classified according to the used approach:

Measurement-based In this approach, parts of the program (or the entire program) are executed on real hardware or simulator using a set of inputs, in order to obtain estimates of execution time.

One option is to perform end-to-end measurement of program execution for the collection of execution time distributions, which are then processed statistically.

A second option is to measure basic block times, such as those described in the program's CFG. Subsequently, the times obtained are combined by some execution time limit calculation technique. Measurement techniques can, in some ways, be used to replace processor behavioral analysis, and can be implemented with code instrumentation or hardware devices for information gathering.

One of the disadvantages of this type of technique is the lack of precision or uncertainty of the results. However, such techniques may be useful for validating static methods of analysis, or tracing the task execution profile in real-time systems.

Static analysis: There are still techniques based on static analysis, which solve the problem of obtaining the worst-case execution time using analytical modeling and static code analysis. These techniques are divided into two parts: *processor behavior analysis* and *the worst path search*. Static analysis may rely on other types of analyzes, many of them based on compiler theory, such as *value analysis*.

Hybrid approaches: Hybrid approaches combine concepts of techniques based on measurement and static analysis. Hybrid approaches consist of identifying unique viable paths (*single feasible path - SFP*). Unique viable paths are paths made up of basic blocks whose executions are invariant to input data. Such paths are identified by static analysis. Subsequently, the execution time of the SFPs are measured in real hardware or in simulators with cycle-accurate precision. The last step is to combine the result of the measurements to obtain the worst path. Safety margins can also be added to the end result, however hybrid techniques suffer from the same problems of measurement-based techniques, although they are more reliable.

3.3 STATIC APPROACH FOR WCET ESTIMATION

Static analysis covers techniques whose reliability can be assured in the context of real-time critical systems. In the following sections we will show the typical phases, variations and possible implementation strategies.

3.3.1 Value analysis

Value analysis has the purpose of calculating the intervals of values that each register can have in each point of the program, represented by the CFG. These values can be used to define memory access address ranges or even automatic extraction of loop iteration bounds. Another use for value analysis is the detection of non-viable execution paths in CFG. For example, this analysis may conclude, in certain situations, that a branch will never be taken, since the register used to evaluate the deviation will always have a value equivalent to false. Such a path can be safely ignored in the worst-case path search. The value analysis can be done through data flow analysis techniques, as described in Chapter 2.

3.3.2 Processor behavior analysis

The processor behavior analysis aims to estimate the execution time of each component of the program. This analysis phase is required to deal with the processor components that cause programs to have context-dependent execution. The time of a particular instruction is affected by the execution history, that is, the execution of previous instructions. As mentioned earlier, several components may affect the behavior of the program over the execution history, such as buses, ***cache***, state of ***pipeline***. The behavior of the processor over all possible execution sequences must be analyzed. The complexity of the architecture influences the precision and the analysis techniques adopted. There are basically 3 approaches to estimate the execution time of instruction sequences in hardware:

Abstract interpretation: abstract interpretation is not a new technique (COUSOT; COUSOT, 1977). This technique is based on the representation of the computations of a program using abstract values from some other descriptive universe. Consider the example of the rule of the signs (COUSOT; COUSOT, 1977): the operation -1515×17 can be described in the abstract universe $\{(+), (-), (\pm)\}$, where the semantics of operations are defined by the rule of signals. The abstract execution of the operation $-1515 \times 17 \rightarrow -(+) \times (+) \rightarrow (-) \times (+) \rightarrow (-)$ proves that -1515×17 is a negative number. Abstract interpretation is always concerned with a specific structure of the universe of computations, as the signal in the case of the previous example. Thus, abstract interpretation is a form of interpretation in which ***value descriptors*** or ***abstract values*** are used instead of concrete values. Abstract values guarantee termination of analysis and describe execution for all possible inputs.

This structure description of computations can be extrapolated to the processor behavior domain, exactly what is done by works such as (SCHNEIDER; FERDINAND, 1999) and (THESING, 2004). These works are based on the construction of an abstract processor model. The complexity of the model depends on the type of target processor that will be used. Simple 8-bit or 16-bit processors tend to have equally simple models. Advanced 16-bit or 32-bit scalar processors with pipelines and cache have complex models, but analyzes of different processor components can be done separately as there are no temporal anomalies. Advanced superscalar processors with performance enhancement features such as branch predictors and out-of-order execution have extremely complex models, and analysis tends to be less modular than in the previous case. For complex processors, this may be the only possible approach. Abstract interpretation of programs considering complex processors can generate node explosion in the CFG, due to the need to represent significantly different abstract states for the same basic block or path, due to highly context-dependent execution. An advantage of abstract interpretation is the ability to analyze complex architectures. One drawback is the complexity of the model, which can be extremely difficult to build. This type of technique can be used in any type of architecture.

Basic block simulation: basic block simulation is an analysis technique that uses an architecture simulator prepared for this purpose. Simulation as a measurement technique that uses conventional simulators combined with specific inputs does not fall into the category described here. Basic block simulation can provide reliable results for compositional architectures. In (ENGBLOM; ERMEDAHL, 1999) and (ENGBLOM, 2002) a technique is presented to obtain execution times in compositional architectures with *pipeline*. In this technique, basic block sequences are simulated for the purpose of obtaining worse individual times. Sequences are used because, in some cases, basic blocks may suffer interference from other blocks, so the worst time is only obtained with the simulation of adjacent blocks. The final time composition considers a *pipeline* compensation factor (δ), since the execution of isolated blocks differs from the sequential execution within the *pipeline*.

Pipeline diagrams/reservation tables: for scalar pipelines, we can use pipeline diagrams as described by (HEALY et al., 1999). This technique constructs tables that represent the occupation of the *pipeline*, taking into account *hazards* and *stalls*. This technique can be used with cache analysis. Similarly, reservation tables are used to compute the time of adjacent basic blocks considering the impact of the resource reservation of the *pipeline* by the instructions. For both techniques, a detailed *pipeline* model must be provided to the WCET analyzer.

3.3.3 Worst-case calculation or path search

When the worst-case is known for each basic block that makes up the CFG of a program, in terms of *cache* and *pipeline* (time), we must search for the worst execution path, or calculate the WCET itself. Such search should consider iteration bounds of loops and possible divergent paths (*if-then-else*) of program flows. To solve this problem, several techniques have been proposed, which use different strategies. According to (WILHELM et al., 2008), the techniques are classified into:

Structure-based calculation : This technique, described in (COLIN; PUAUT, 2000), computes an upper bound of the execution time through a *bottom-up* traversal of the program syntax tree. The syntax tree, as its name says, describes the program in a high-level structure (language level), where the basic blocks, which are the nodes of the CFG itself, are placed on the leafs.

This traversal of the syntax tree works by combining computed loop iteration bounds for expressions and uses composition rules to reduce them. In this way, the program's syntax tree is reduced until it reaches a single node. In the process of reducing and collapsing the nodes, their times are also combined, so that at the end of the process, there is only one node in the syntax tree, along with its respective WCET.

According to (WILHELM et al., 2008) this technique has some problems. The first problem is inexpressiveness, since not every flow of control can be represented, only simple structures. The second problem is optimizations, which cause the generated code to lose its match with the program's syntax tree. As an example, we consider the Figure 10. Subfigure 10(a) shows a CFG with execution times annotated in the nodes, and a loop with iteration limit of 100. The application of the structure-based technique is shown in Figure 10(d).

Path-based calculation : In the path-based approach ((HEALY; WHALEY, 1999), (STAPPERT; ALTENBERND, 2000), (STAPPERT et al., 2001)), a WCET upper bound is computed composing limits of several sub-paths of the CFG, searching the worst total path. In this technique, paths are represented explicitly, so the number of paths grows exponentially with the number of branches, as in nested loops for example. This explosion of paths can be addressed with heuristic search for example. Figure 10(b) shows an example of application of this technique.

Based on implicit path enumeration technique : The technique that uses implicit path enumeration (IPET calculation) was proposed by (LI; MALIK, 1995) and expanded to more complex models by (LI et al., 1996), (ENGBLOM, 2002), (THEILING, 2002) and (ERMEDAHL, 2003). This technique consists of modeling the CFG of the program as a set of linear inequalities. Such inequalities describe the flow of the program in terms of execution constraints and execution times of each basic block. In this technique, each basic block and each edge of the CFG is assigned a coefficient t_{entity} which represents the contribution of this component over the total time for each time it is executed. The number of times a component is executed is denoted by x_{entity}.

For this problem, the solution is obtained by integer linear programming, where the objective function is the maximization of the flow in the CFG, that is, the flow that generates the WCET. In other words the maximization the sum $\sum_{\forall i \in entities} x_i \times t_i$. Currently, IPET is the most used technique by existing analyzers because of its efficiency. As an example of application of this technique, Figure 10(c) is considered.

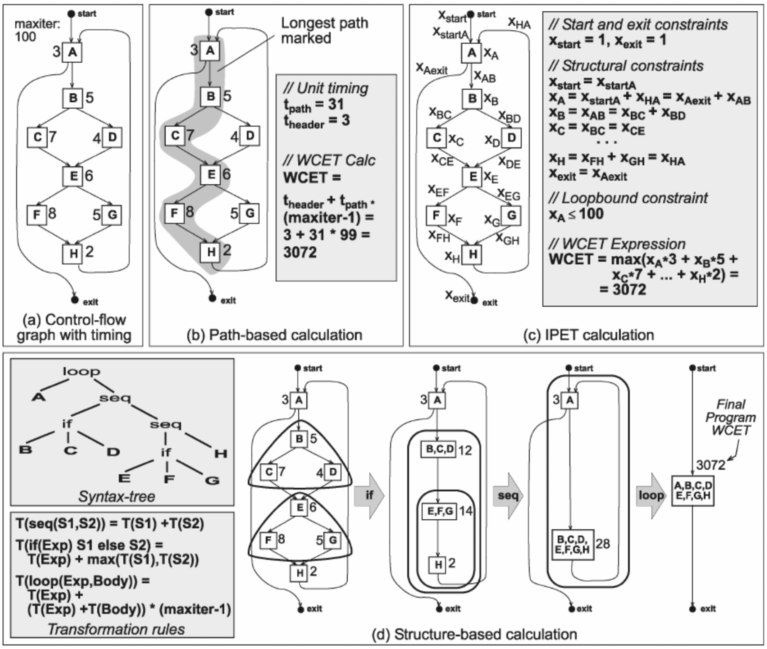


Figure 10 – Techniques to WCET search/calculation, obtained from (ERMEDAHL, 2003)

3.4 CHAPTER SUMMARY

We presented in this chapter concepts related to obtaining the WCET of programs and their complicators. This obtainment must consider elements of the architecture used, since these impact significantly on the choice of techniques and their effectiveness. If the processor used has temporal anomalies, the analysis may not be possible due to the explosion of states in the calculation process or even produce useless results.

Basically, WCET analysis must perform the processor behavior analysis considering all its elements, such as caches and branch predictors, and the subsequent search for the worst possible path. The behavior of the processor can be analyzed with abstract interpretation, resource reservation tables or even basic block simulation. When the processor is sufficiently complex, abstract interpretation may be the only alternative. When the processor has no temporal anomalies, compositional techniques can be used. Compositional techniques are interesting because they do not generate explosion on the search space of the analysis, necessary to represent all possible contexts of each of the basic blocks. This explosion induces a considerable increase in the time taken to obtain the solution. In relation to the tools, there are alternatives (commercial or not) such as AiT[1], RapiTime[2], BoundT[3], Sweet[4] and Otawa[5].

[1] http://www.absint.com/ait/
[2] http://www.rapitasystems.com/products/rapitime
[3] http://www.bound-t.com/
[4] http://www.mrtc.mdh.se/projects/wcet/
[5] http://www.irit.fr/recherches/ARCHI/MARCH/OTAWA/doku.php

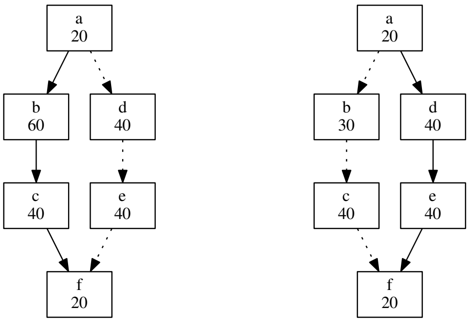
4 OPTIMIZATIONS FOR WCET REDUCTION

The objective of this chapter is to present the state-of-the-art in optimizations for *worst-case execution time* reduction. Traditional optimizations are often targeted to different goals, such as ACET (*average-case execution time*) reduction, energy saving and code size reduction. To be able to generate and optimize code aiming at WCET reduction, a compiler must have an integrated analyzer capable of performing temporal estimates about the program (ZHAO et al., 2006) (LOKUCIEJEWSKI; MARWEDEL, 2011), or be supplied with some kind of information that complies with this objective. Otherwise, the compiler would have difficulty in finding possible optimization potentials that actually affect the WCET of the program. To have an effect on the WCET of a program, an optimization must take into account the existence of the WCEP, or *worst-case execution path*, which is the execution path in the CFG that generates the WCET, when executed.

According to (LOKUCIEJEWSKI; MARWEDEL, 2011), if we want to reduce WCET, it is not appropriate to conduct all desired optimizations looking to a single initial WCET calculation. The reason is the instability of the WCEP, which can change at every optimization application. Figure 11 shows an example of WCEP instability. In this example, the numbers represent the worst-case times of the basic blocks, while solid edges represent the WCEP. In the original version (Figure 11a), we have a WCET of 140 cycles, which is related to the path $a \rightarrow b \rightarrow c \rightarrow f$. The reduction of 30 cycles of the original WCET, however, decreased only 20 cycles of the WCET, because the WCEP was changed to $a \rightarrow d \rightarrow e \rightarrow f$.

We can observe in the previous example that any optimization that changes the WCEP can change the WCET. The reduction of x cycles in the WCEP does not imply in a reduction of the WCET in x cycles.

Compiler optimizations designed to WCET reduction, as well as any kind of optimization, must meet the following general objectives (AHO et al., 2008):



(a) Original code (b) After optimization of basic block b

Figure 11 – Example showing a WCEP switch after the application of an optimization

- They must be correct, ie, preserve the semantics of the compiled program: the correctness is of extreme importance because a compiler that generates faster, but incorrect code is useless.
- They should improve the performance of many programs: optimizations should be effective in performance improvement for many of compiled programs. For example, performance can be understood as execution speed or in the case of embedded systems, size of the generated code. As this work focuses on real-time systems, performance increase can be characterized also as WCET reduction.
- They must keep the compilation time reasonable: to support rapid development and debugging cycles, the compilation/optimization should be performed with the shortest time possible. This objective is partly easy to achieve, since today's computers become faster and faster. Optimizations for real time tend to be extremely slow because the need of invoking external WCET analyzers, which are in turn generally slow.

- They must require manageable engineering effort: compilers are extremely complex software. Maintenance costs of any internal component must be appropriate. Many optimizations can certainly be implemented with great effort, but we should only consider those that offer benefits for a reasonable amount of programs.

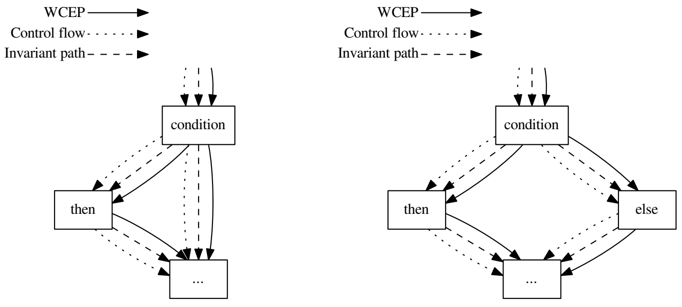
According to (LOKUCIEJEWSKI; MARWEDEL, 2011), reducing WCET in compilers is difficult by the following reasons:

- There is a need for a timing model of the target processor or equivalent data must be supplied by some WCET external tool.
- The need of new optimization paradigms, which are not targeted to ACET, as these are not suitable for WCET.
- Any code transformation must be aware of WCEP switches.

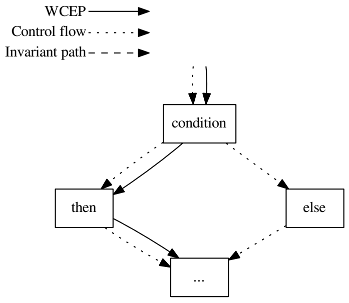
4.1 DEALING WITH WCEP SWITCHES

If the compiler is aware of possible WCEP changes, it can choose two options when applying an optimization: try to find out the new WCEP, or ignore it and continue exploring the initial estimate, with the risk of optimizing paths not beneficial to WCET. If the compiler chooses to get the updated version of WCEP at every optimization application, the total compilation time may be impractical for complex applications due to successive invocations to the analyzer.

In (LOKUCIEJEWSKI et al., 2009) is presented a technique called *Accelerating Optimization by the Invariant Path*, which is more easily applied to representations at source code level. This technique relies on the fact that, in some cases, may be unnecessary to perform the calculation of the WCET and its corresponding WCEP after some optimizations as this can not have been changed. The WCET calculation can be avoided when optimizations are applied to basic blocks belonging to the *invariant path*. The invariant path is a subpath of the WCEP that is always in the WCEP, regardless of the changes applied to the program being compiled. The consideration of the invariant path can accelerate the application of optimizations for WCET reduction, since it reduces the time needed for analysis. All linear path that is contained in WCEP, and not in a mutually exclusive subpath is considered an invariant path. Programs with CFGs containing divergent flows and mutually exclusive subpaths represent a challenge in the identification of invariant paths, for both compiler and for WCET analyzer.



(a) Invariance in an if-then sentence (b) Invariance in an if-then-else structure



(c) Possible WCEP switch

Figure 12 – Scenarios of invariances and possible WCEP switch

Invariance scenarios are shown in Figure 12, considering structures of the ANSI C language:

***If-then* invariance:** The *if-then* branch structure shown in Figure 12a is a conditional execution sentence. Depending on the condition, either the path containing the conditional then block executes, or the path that deviates the *then* block executes. In this case, either the WCEP pass through the *then* block or the *then* does not contribute to the WCET. In terms of invariant path, the WCEP going through a conditional *then* is also part of the invariant path, this because the other viable path of the *if* (*else* block) sentence does not contain code that can become the new WCEP.

When we use context sensitive WCET analysis, there are chances of the WCEP to pass through both paths in different contexts, as shown in Figure 12a. In this case, optimizations applied to the conditional *then* certainly do not alter the WCEP.

***If-then-else* invariance:** In case of the *if-then-else* structure (Figure 12b), sensitive analysis context can determine that both the *then* block and the *else* block are traversed in different situations. This means that both blocks always contribute to the WCET, then both can be declared as included in the invariant path. Optimizations can be safely applied to the two paths without the need of WCEP recalculations. WCET analysis can detect that the condition of the *if-then-else* structure will always be evaluated as *true* or *false*. As a result, one of the paths will never be executed and may be declared as *dead* subpath, converting the *if-then-else* structure in a simple *if-then* structure, allowing the classification of the remaining path as invariant.

Non invariance: When the analyzer can not statically infer which path will be taken in a *if-then-else* structure (Figure 12c), any mutually exclusive path could become part of the WCEP. In this case, an update of WCEP at every optimization application is necessary.

Another approach consists in performing an approximated WCET recalculation using ILP (*integer linear programming*) inside the compiler. This alternative is feasible, but it is only an approximation, not replacing the traditional analysis of WCET. Such an approach is achieved by using initial data obtained in a traditional WCET analysis, reapplied in a simplified model used by the compiler.

4.2 CHARACTERISTICS EXPLOITED BY OTIMIZATIONS

Regarding the computer architecture, optimizations can explore different features, such as:

Parallelism: current processors often exploit instruction level parallelism. This parallelism can appear in two ways. The first way is transparent to the programmer. In this strategy, the program is written to be executed in a fully sequential manner, where the hardware checks at runtime dependencies between instructions, and issues these in parallel whenever possible. Nevertheless, the compiler can rearrange instructions to help the hardware in the issuing work.

The second form of parallelism appears explicitly in the instruction set architecture (ISA). In this alternative, also known as VLIW (*very long instruction word*), each machine instruction may issue multiple operations in parallel. In this case, the compiler is responsible to schedule independent operations in a same instruction.

Memory Hierarchies: a memory hierarchy consists of multiple levels of memory, with the faster (but smaller) being the closest to the processor. Generally, fast memories operate as *caches* of the slowest. In a program, the average access time to the memory is reduced when most accesses are satisfied by the fast memory.

4.3 OTIMIZATIONS TO REDUCE WCET

Optimization involving WCET reduction can be applied in different internal representations of a compiler, reflecting different stages of compilation. Some techniques work at source code level, others at assembly level or in the organization of the final code (*layout*). The following representative techniques existing in the literature will be presented according to this classification.

4.3.1 At source code level

Source code level optimizations are important because they act in the high-level representation of compilers, ie they are portable. This type of optimization also increases the opportunities for further optimizations in the compiler, such as those that occur in lower level representations. A requirement to perform these type of optimizations is the use of *back-annotations*. Back-annotations transform WCET information regarding low-level intermediate representations in equivalent information, but mapped to high-level representations. In the literature, it is reported that few compilers implement back-annotations of WCET information, as TUBOUND(PRANTL et al., 2008) and WCC(FALK et al., 2006). The following optimizations exist at source code level to reduce WCET:

Procedure cloning: Also known as specialization of functions (LOKUCIEJEWSKI; FALK, 2008), it is typically applied to source code level and is widely known in ACET reduction context. Many real-time applications have their executions highly dependent on input data and parameters. Procedures may have internal loops whose iteration counts are affected/defined by call parameters. When this is the case, WCET analyzers generally consider the worst possible parameter as iteration bound, which may be pessimistic for a wide range of uses of the considered function. WCET analyzers can also perform context sensitive analysis, but this often affect the complexity of the analysis, leading to the explosion of states in case of nested loops. The procedure cloning allows the creation of specialized copies, ie, with statically defined parameters for each of the callers procedures, facilitating and increasing the accuracy of the analysis. Increased precision is commonly observed as consequence of:

- More accurate definitions of iteration bounds of loops due to the replacement of parameters by constants in the cloned version;
- Elimination of possible infeasible paths. The use of constants in conditional expressions allows the identification and removal of infeasible paths;
- Often, cloned versions do not generate functions with lower WCET but opens chances for future cloning.

There are different strategies to select functions/procedures as candidates for optimization. Some parameters must be considered: definition of the maximum size of the function to be cloned. Another parameter to be considered is relative to the restrictions on the occurrence of constants as function arguments. For example, we may consider only cloning functions whose calls involving a given constant as parameter represent at least a half of total calls of this function. If the considered frequency is not met, then the function is discarded for cloning (considering the parameter chosen).

Superblock Optimizations (1): The application of optimizations often fails to exploit all the potential opportunities in a program due to the limitations imposed by basic block boundaries. To overcome this problem, we can use a superblock structure, which has already been explained in Chapter 2. A superblock contains several basic blocks, and allows the application of optimizations that cross the borders of these. This technique is already used to minimize ACET. For superblocks formation, basic block execution counts are required. In the case of ACET minimization, such information can be obtained through *profiling*. For real-time systems, the formation of superblocks should be guided by WCET information. Considering the optimization presented in (LOKUCIEJEWSKI et al., 2010), there is still the difference (in relation to the traditional superblock formation) that superblocks are formed at source level (WCET-aware source code superblocks), not at assembly level. Such change allows an initial code restructuring that can expose more opportunities for future optimizations.

In order to select the trace to form a superblock in a control flow graph, we can use weights of the nodes (basic block execution count) or edges (flow transfer counts), with the last generating the best results. Starting from the CFG, we must select the *trace* that maximizes the sum of the weights (nodes or edges), and that has not been previously selected. After the selection, we proceed with the operations necessary to form a superblock.

The (LOKUCIEJEWSKI et al., 2010) technique uses IPET to ensure that subsequent optimizations do not operate on an out-of-date WCEP. The initial IPET information is obtained from an initial invocation to a full-featured analyzer. The IPET estimates are less accurate, but they also are less expensive than invoking the analyzer at each trace selection.

Using the superblock, optimizations such as *elimination of common subexpressions* and *elimination of dead code* are applied. As a result, WCET is reduced for many cases of the benchmark suite utilized.

Loop unrolling (1): loops always have good optimization potentials in modern architectures. The loop unrolling technique has been used as an effective strategy to improve the average performance of programs. This technique consists of replicating the loop body a few times, inserting extra code to verify the stay/exit conditions, when necessary. The number of replications is called *unrolling factor u*, and the loop in which optimizations are applied is commonly called *rolled loop*. The benefits of the technique are:

- Reducing overhead of increment operations and condition tests;
- Increase instruction level parallelism, which in turn enables other types of optimizations.

However, some negative effects can be observed if the technique is not carefully applied:

- Impact on the instruction *cache* due to resulting code increase;
- Increase of the code required for register saving operations (*spill*), due to the increased pressure on them.

The technique, when applied in its traditional form may not produce WCET gains. In (LOKUCIEJEWSKI; MARWEDEL, 2010) they present a technique that considers the benefits for WCET by evaluating each loop individually as well as the possible negative effects. The technique is applied at source level to enable possible future optimizations. The essential point of the technique is which *unrolling factor* must be used for each loop, which depends on the following parameters:

1. Iteration limits for each loop. Using context-sensitive techniques, it is possible to obtain several bounds;
2. Constraints about the available memory for the program and *cache* parameters;
3. Estimating the amount of generated *spill* code.

Then, the values considered for *rolling factor* should:

- Be between the *least common prime factor* (LCPF) among all context-sensitive iteration bounds and 1. This is done to avoid unnecessary branches in loops. Then, the unrolling factor must be such that the stay/exit conditions must be checked only once at each iteration. This is due to the fact

```
for(i = 0; i < 100; i++){
    x[i] = x[i] + y[i];
    if(w){
        y[i] = y[i] * 2;
    }else{
        y[i] = 1;
    }
}
```

(a)

```
if(w){
    for(i = 0; i < 100; i++){
        x[i] = x[i] + y[i];
         y[i] = y[i] * 2;
    }
}else{
    for(i = 0; i < 100; i++){
        x[i] = x[i] + y[i];
        y[i] = 1;
    }
}
```

(b)

Figure 13 – Example of loop unswitching. (a) before, (b) after

that branches introduce possible degradations in the *pipeline* and inaccuracies in the WCET analysis. Note that the value should be as large as possible between 1 and the LCPF of the obtained bounds, which satisfies the following restrictions;

- Should cause the loop to not exceed the available memory;
- Should make the loop to fit into *cache*.

Considering the unrolling factors obtained, the most promising loops are optimized, that is, the ones with the best benefits with respect to WCET and code size.

Loop unswitching: loop unswitching optimization is a well-known transformation used for ACET reduction that can also be applied for WCET reduction (LOKUCIEJEWSKI et al., 2009). This transformation consists of moving out of the loop conditions that are invariant. In the case of if-then-else sentences, the body of the loop is replicated within the block then and else. The benefits of this transformation are reducing the number of branches, improving pipeline performance, and exposing more opportunities for loop parallelization. An example of the application of the technique can be seen in Figure 13.

Like loop unrolling, this technique can not be applied extensively across all candidate loops if there are memory constraints. In this case, we should also consider a trade-off between execution time gains (ACET or WCET, depending on the goal) and code increase.

The technique for WCET reduction consists of:

- Perform a WCET analysis to obtain data such as condition execution counts and WCEP, as well as to obtain information about the available memory for the program;
- Perform the selection of all candidate loops, including those outside the WCEP. This is done because a loop could become a member of the WCEP after possible WCEP switch;
- Optimize the loops according to the order: loops whose execution frequencies of the invariant condition are larger (WCET analysis) are optimized first. In case of a tie, we consider the branch with the highest WCET. In the case of a WCET tie, the branch with the least amount of code is chosen. For each loop optimization, the available memory must be considered, and in the end, the WCEP must be recalculated due to possible path changes. The notion of invariant path can be useful to avoid possible unnecessary calculations of WCET/WCEP.

Function inlining: The decision to inline a function is usually guided by compiler heuristics. It is common to consider the size of the called function in statements or generated instructions. Then we compare his size with a certain threshold, and we can discard the possibility of *inlining* if it exceeds such value. Generally, these type of heuristics tend to be very conservative, operating only on very small functions to avoid effects such as excessive code expansion.

In (LOKUCIEJEWSKI; MARWEDEL, 2011) it is shown that the application of a simple function selection heuristic ends up degrading the WCET in a selected set of applications. Then, that work proposes the use of *machine learning* based heuristics to reduce WCET by *function inlining*.

4.3.2 At assembly level

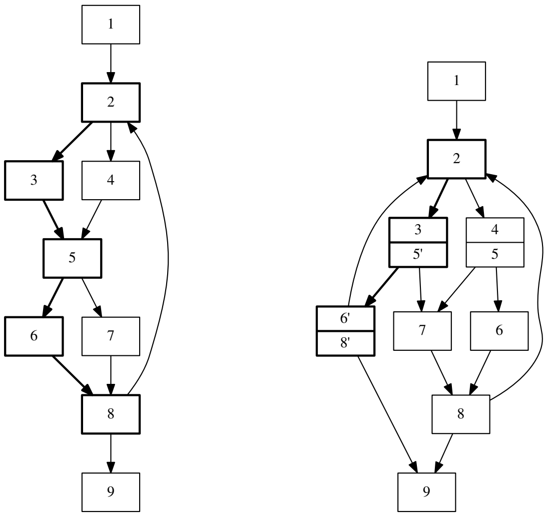
Assembly-level optimizations operate directly in the machine language of the target processor. These optimizations are applied on the final stages of compilation, since the code generation has already been achieved.

Superblock formation (2): The formation of superblocks at the assembly level for optimizations is explored by (ZHAO et al., 2006). WCET information is generated by the compiler with the purpose of selecting the *trace* that will originate the superblock. An example of superblock formation from WCEP information is shown in Figure 14. In Figure 14a the blocks and transitions that compose the WCET are highlighted, while Figure 14b presents the superblock formed for this example.

With a superblock, we can, for example, eliminate flow transfers by grouping blocks wherever possible. After the formation of the superblock, other optimizations can be applied, such as the following ones.

Path duplication: The path duplication technique (ZHAO et al., 2006) is useful for architectures where flow transfer is considered costly. This technique consists of duplicating the path (included in the WCEP) in a certain loop, after the formation of the superblock. An example can be seen in Figure 15, which represents the application of this optimization on the example of Figure 14b. In this example, it is possible to significantly reduce flow transfers, for successive loop iterations.

***Loop Unrolling (2)*:** In this technique (ZHAO et al., 2006), applied to the program at assembly-level, the entire loop is duplicated, not



(a) Before the superblock formation

(b) Superblock formed

Figure 14 – Example illustrating the superblock formation

just the WCEP components inside it, like occurs in the path duplication. In (ZHAO et al., 2006) only innermost loops are considered, and the *unrolling factor* is set to 2 to limit code increase. An example of the *loop unrolling* application of (ZHAO et al., 2006) can be seen in Figure 16. In Figure 16a the loop is unrolled twice, highlighting the components of the WCEP. Finally, the formation of the superblock for the example of Figure 16a is shown in Figure 16b using the blocks that are components of the WCEP. Experimental results indicate that *path duplication* results in less code increase, however *loop unrolling* has a better WCET reduction. Experiments using a processor with no caches showed that a WCET reduction of up to 10% was achieved for all benchmarks.

Trace scheduling: *Trace scheduling* has already been presented in the

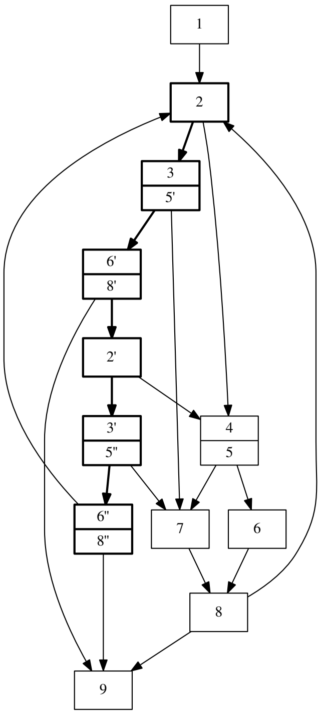
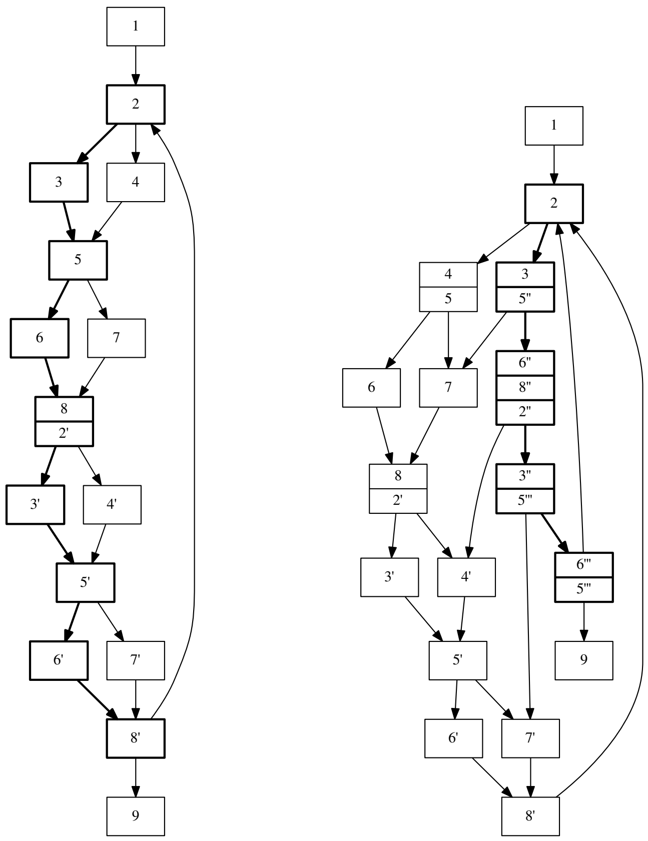


Figure 15 – Example illustrating the path duplication technique.

compilation chapter. Here it is a variant of the technique that considers WCET information for *trace* selection (LOKUCIEJEWSKI; MARWEDEL, 2011). The idea of the algorithm is to use the worst path obtained in the WCET analysis to select the *trace*, using parameters such as maximum number of blocks. The remaining steps follow the traditional technique, which involves the construction of the dependency graph and *list scheduling*. The process is repeated while there are non-scheduled blocks or a timeout expires. The WCET information is also updated frequently so that the algorithm always selects the *trace* that most



(a) Application of *loop unrolling* in the example of Figure 14a

(b) Superblock formation for the example of Figure 16a

Figure 16 – Example of loop unrolling optimization and superblock formation

impacts on the WCET. At the end of the process, if the original program still gets the best WCET, then the optimized version will be discarded by a *rollback* mechanism. As a result, WCET is reduced for most benchmarks.

Through static branch prediction: dynamic branch predictors are used to improve program performance by reducing idle cycles that this type of operation can generate. Dynamic predictors are usually accurate if a branch is to be taken. However, they impose difficulties in obtaining WCET, since they depend on execution history. On the other hand, static predictors are inherently deterministic from the WCET analysis point of view. Such mechanisms can be configured considering that the branch will always be taken, or that it will never be taken (*branch not taken*, or *fallthrough*). Some architectures allow we to choose the direction of the branch, through specific instructions. In these architectures, the compiler can give compile-time suggestions about the behavior of each branch. These suggestions can be given based on profile information (FISHER; FREUDENBERGER, 1992) or directly by static analysis (PATTERSON, 1995). However, these alternatives consider only the average execution case, and are not suitable for WCET reduction.

When the predictor can be handled by the compiler, it hints the direction of each branch, like the technique of (BODIN; PUAUT, 2005). The basic principle of the algorithm is to statically predict all conditional branches that appear along the WCEP (*Worst-case execution path*, i.e., the path that generates the worst-case execution time when traversed in a program execution), considering possible path changes. This approach can be used in processors with support for compiler-directed branch prediction. The approach is modeled by Algorithm 1 and it uses the following notation: CFG represents the program control flow graph, which is supposed to be known at compile time. The nodes of a CFG are basic blocks (*BB*). Basic blocks are sequential instructions and they can end at a control flow instruction and begin at a control flow target. A basic block may end at an arithmetic instruction if the next instruction in memory is a branch target. If a basic block ends with a conditional branch, then the function *is_conditional_branch* returns true. When such a branch exists in a basic block, $BB.tk$ and $BB.ft$ represent a taken target and the fall-through basic blocks respectively. The taken target basic block is the next basic block when a branch is actually taken and the fall-through is the "adjacent address" basic block after the branch instruction when the branch is not taken. Each basic block can be in one of two states: $BB.predicted = true$ or $BB.predicted = false$.

If a basic block containing a branch instruction is not predicted, both successors $BB.tk$ and $BB.ft$ are considered mispredicted, implying a penalty for both of them when calculating the WCET. This penalty for both branch targets depends on a change on how the WCET is calculated by the analyzer, because this relates to a pessimistic behavior that simulates a branch with no direction (or prediction). This pessimistic behavior is not a common feature present in WCET analyzers, so, a custom implementation may be necessary. Another requirement is a worst-case execution count for each basic block. Fortunately this information is commonly provided by WCET analyzers.

If a basic block is considered predicted ($BB.predicted = true$), then we have a predicted direction for it, which can be *taken* or *fall-through*. The entire procedure relative to WCET analysis is abstracted by function $estimate\_WCET$. The result of a WCET analysis is the worst-case execution path (WCEP) annotated with the worst-case execution count for each basic block, which is denoted by $count$. Basic blocks with higher count values execute more times than basic blocks with lower count values in the worst-case execution. For a more detailed description of this technique, see (BODIN; PUAUT, 2005). We will call this technique *classic approach*.

The work of (BURGUIERE et al., 2005) proposes the use of a CFG structure to statically predict branches. This technique requires a well-structured program, e.g., all loops have only one exit, and a branch is *taken* to exit from it. Branches inside loops

Algorithm 1 Algorithm for static branch prediction (BODIN; PUAUT, 2005).

```
 1: procedure SET_PREDICTIONS($CFG$) ▷ Set predictions along the WCEP
 2:     bool $converged \leftarrow false$
 3:     int $dir \leftarrow undefined$
 4:     for all $BB \in CFG$ do
 5:         $BB.predicted = false$
 6:     end for
 7: {Step 1: WCET estimation}
 8:     $WCEP = estimate\_WCET(CFG)$
 9:     while $converged = false$ do
10: {Step 2: Issue static branch predictions along the WCEP}
11:         for all $BB \in WCEP$ do
12:             if $is\_conditional\_branch(BB)$ then
13:                 if $BB.ft \in WCEP \wedge BB.tk \in WCEP$ then
14:                     if $count(BB.tk) \geq count(BB.ft)$ then
15:                         $dir \leftarrow taken$
16:                     else
17:                         $dir \leftarrow fall-through$
18:                     end if
19:                     if $BB.predicted = false$ then
20:                         $BB.predicted \leftarrow true$
21:                         $BB.direction \leftarrow dir$
22:                     end if
23:                 else
24:                     if $BB.tk \in WCEP$ then
25:                         $dir \leftarrow taken$
26:                     else
27:                         $dir \leftarrow fall-through$
28:                     end if
29:                     if $BB.predicted = false$ then
30:                         $BB.predicted \leftarrow true$
31:                         $BB.direction \leftarrow dir$
32:                     end if
33:                 end if
34:             else
35:                 $BB.predicted \leftarrow true$
36:             end if
37:         end for
38:     {Step 3: WCET estimation}
39:         $WCEP \leftarrow estimate\_WCET(CFG)$
40:         if $\forall BB \in WCEP, \quad BB.predicted = true$ then
41:             $converged \leftarrow true$
42:         end if
43:     end while
44: end procedure
```

must be predicted as *not taken*. With this prediction, if a loop iterates n times, then we have n correct predictions and 1 misprediction at the end of the execution. If a loop is repeated m times, then we have $m \times n$ correct predictions and m mispredictions. For conditional structures, the prediction should be to the path that has the longest execution when the branch is mispredicted. To do this, an analyzer that can process parts of a program is necessary. The algorithm predicts branches in a bottom-up way over a control tree produced by the structural analysis.

Through Register Allocation: The next alternative for WCET reduction presented in this chapter refers to register allocation strategies. Graph-based algorithms for register allocation employ simple heuristics to decide which registers will be stored in memory (*spill*). To reduce WCET, it is necessary for compilers to have some knowledge about the timing model of the target architecture, in order to effectively allocate registers for WCET reduction.

In (FALK, 2009) the traditional graph coloring algorithm is extended for WCET reduction. Since the allocation of registers requires WCET information and a program can only be executed or analyzed after the allocation of registers, there is a problem of mutual dependence. This problem is solved by calculating the WCET of the program assuming pessimistically that all virtual registers or variable of the program will suffer *spill*. This initial calculation and its respective WCEP serve as input to the graph coloring algorithm proposed in the paper. The register selection heuristic operates by trying to reduce the amount of *spill* code generated along the worst possible path, which is recalculated at regular intervals to address the worst path instability problem. The proposed technique can significantly reduce the WCET of programs when compared to the traditional graph coloring technique.

In (FALK et al., 2011) is presented a technique that offers better results than the one based on graph coloring. This technique, based on integer linear programming, models the costs of *spill* operations by considering the execution of these in the *pipeline*, and *worst-case execution time* is also modeled. The result is obtained by the traditional ILP register allocation technique combined with the modeled constraints.

Register allocation together with instruction scheduling considering WCET is the proposal of (HUANG et al., 2012). In this work, they consider clustered VLIW architectures. In such architectures, registers and functional units are grouped into *clusters*, where functional units in a same *cluster* have access to a same subset of registers. This is done to reduce processor bottlenecks, reducing propagation and communication delays. The technique of (HUANG et al., 2012) consists of first performing a static analysis of WCET on a program *P*, subjected to aggressive scheduling and a *cluster* assignment without considering registers pressure issues[1]. Then, a node representing a basic block is selected for register allocation, where the rescheduling of instructions is taken into account. For the instructions of the selected basic block, instructions rescheduling and *cluster* assignment are done during the register allocation, in order to reduce the execution of expensive *spills*. At the end of the process, the WCEP is updated and the next node is selected, and the process is repeated until all nodes have been selected.

4.3.3 Through code layout

Another way to reduce program WCET is by reorganizing/repositioning its code. Techniques that work by altering the code layout are intended to improve the instruction's locality in cache access, thereby reducing cache misses and jump penalties. Traditional code positioning optimizations can use profile information to find the best ordering of basic blocks. However, programs' profiles refer to the execution of the code in their ACET and are not useful for improving WCET. Ba-

[1] Pressure on Registers occurs when there are fewer available registers than the optimal number of these, inducing a greater number of *spill* and *reload* operations.

sic block positioning techniques and procedures should be based on strategies related to the worst flow of the program.

Positioning to improve branch performance: In (ZHAO et al., 2005) is presented the first proposed technique for code positioning at the basic block level based on worst-case information. In this work, a WCET analyzer coupled to a compiler is used. The analyzer provides information about the worst paths so that the compiler can organize the code in order to reduce branch penalties. The architecture used is the StarCore SC100, a DSP with different branch and jump penalties.

In this work, a path-based analyzer is coupled to the compiler. The paper argues that structure-based schemas are not suitable for code-positioning because they reflect the high-level structure of the program, and any optimizations cause one-to-one matching between the compiled code and the high-level program to become invalid. It is also said that IPET-based approaches, although simple, consume time for solution and the result is represented by a single numerical value (the WCET itself), which can not be used to guide optimizations. The article also discards symbolic execution because it is slow and significantly increases the compilation time.

The goal of this technique is to minimize the execution time of all paths, since any path that has its time increased may be a new candidate for worst path. To accomplish the objectives, the technique tries to discover the best permutation of basic blocks in a function. Brute force approaches are not always feasible, because if there are n blocks, there will be $n!$ permutations, which will increase the compilation time excessively.

The central idea of the positioning algorithm is to first mark each edge of the CFG as "not positioned" and, at each iteration, find the edge of the CFG that most contributes to the WCET and then place the blocks connected by this edge continuously in the memory. This step will potentially generate a new worst path. The next step is to recalculate WCET for all paths, and again choose a new edge for repositioning. The algorithm ends when all the edges have been positioned.

An advantage of the technique is that it often improves the average-case execution time too (ACET). The technique also presents the same results as the brute force algorithm. One of the disadvantages of the technique is compile time since the WCET analyzer must be invoked at each edge positioning. Another disadvantage is that the path analysis of the analyzer itself is not the most efficient technique among those available in the literature.

Block placement to improve cache behavior : Another technique is presented in (FALK; KOTTHAUS, 2011), which seeks to position the basic blocks in memory in order to optimize *cache* behavior and in the worst case. In this work they use the concept of a formal *cache* model, which can capture all current types of cache. This work was the first to consider code positioning to reduce WCET considering cache. The motivation of the technique is that code fragments that are mapped to the same *cache* line and have high temporal locality (execute on the same loop iteration, for example), evict each other from cache. The resulting *cache* faults are called *conflict misses*, because the code fragments conflict if their *cache* lines overlap. This overlapping of cache lines can be resolved by positioning such code fragments continuously into memory, resulting in less conflict.

For block placement, a conflict graph is constructed. This graph $G = (V, E, w)$ is a directed graph where the nodes have weights. A node v_i denotes a basic block. The set of nodes $V = v_1, ..., v_n$ includes a node v_i if this node conflicts with some other node v_j. The set of edges E contains an edge $e_{i,j}$, if a line of *cache* containing code of v_i can be replaced (evicted) by v_j code. The weight $w_{i,j}$ of an edge $e_{i,j}$, approximate the number of cache misses that occur during execution of v_i, which are caused by v_j, since v_j evicts *cache* lines from v_i. The obtainment of the aforementioned graph is done from a WCET analysis, applying the formal cache model, where further refinements are done using control flow analysis, *may* analysis and *layout* analysis of memory.

Positioning heuristics act through the following greedy approach: edge weights are used to identify code fragments with greater potential of cache misses. Starting from the heavier edge, the algorithm evaluates the influence on WCET and the cache conflicts when relocating the basic blocks from the latter to a continuous position. If there is a WCET reduction, the reallocation is maintained and the algorithm continues through the next heavier edge, and so on. This ensures that the WCET of an earlier step will always be worse than that of a later step. As a result, the positioning algorithm is able to reduce WCET in programs compiled for the Infineon TriCore TC1797 processor, using the aiT analyzer for conflict graph generation. Differently from the prior technique, significant improvements in ACET were not obtained, and in some cases there was worsening. This worsening may be explained by the insertion of additional jumps when relocating basic blocks. As in the prior technique, there is a high cost associated with invoking the WCET analyzer at each iteration of the algorithm, where the total time can take many minutes (240 seconds on average on a typical computer) for some benchmark programs used.

Positioning Functions to Improve Cache Previous techniques were concerned with positioning code in a basic block level. There is also an optimization technique that works with procedure or function positioning (LOKUCIEJEWSKI et al., 2008). In the context of traditional systems, this theme has been studied for some time by (GLOY; SMITH, 1999), (GUILLON et al., 2004) and (LIANG; MITRA, 2010), however the proposed techniques consider only reduction of the average execution time, which is not suitable for real-time systems. Finding the optimal position with respect to WCET involves the computation of all possible permutations, which becomes exponential with the number of procedures.

Positioning functions to reduce WCET, or ***WCET-Centric Call Graph-Based Procedure Positioning*** was primarily proposed by (LOKUCIEJEWSKI et al., 2008). The work proposes two optimization algorithms that exploit the memory hierarchy to reduce WCET. WCET reduction is a result of better utilization of instruction cache memory. Both associative caches and directly-mapped caches can benefit from such a technique. The algorithm at its core is similar to traditional procedural positioning, except that it uses annotated procedure call graph with WCET information instead of profile data. There were basically two variations of the algorithm:

Greedy Version: In the greedy approach the algorithm reorders procedures that are more promising for WCET reduction. At each step, the WCET is recalculated before the final commit of the change. At each step, the annotated call graph also needs to be generated together with WCET.

Heuristic Version: A second approach is to position the procedures based on a single instance of WCET and call graph, which are generated based on the original program. This version of the algorithm exploits the continuous positioning of nodes with higher call frequencies, as these will certainly be in the worst-case path. This version of the algorithm gains a lot in processing time compared to the previous one, but does not produce such good results, and in some cases, the WCET of the program may be worsened.

The presented variations obtained, for the benchmarks used, a mean reduction of 10% for the greedy algorithm and 4% for the heuristic version. The authors point out that the techniques can be improved with the use of the procedure cloning technique, which specializes certain functions, whose execution bounds are context dependent. One of the disadvantage of the techniques was the time spent by the algorithms, which reached 183 minutes for one benchmark.

Regarding positioning of functions, the work of (FALK; KOTTHAUS, 2011), which was previously presented, can be used for the positioning of complete functions.

Positioning of functions by COP: (MARREF; BETTS, 2011) proposes another technique aiming at WCET reduction with procedure repositioning. The proposed technique is based on *Constraint-Optimization Problem* (COP). The objective of this technique is the same of the previous one: to position the basic blocks in order to reduce *cache misses*. The article explains how to model the code positioning problem for WCET minimization as a COP problem. It is also presented that the technique never increases the WCET of programs, and can be used in conjunction with other traditional memory positioning techniques. The article does not present real implementations done in compilers, for evaluation using benchmarks.

Incremental function positioning: In (MEZZETTI; VARDANEGA, 2013) they present a technique that proposes a fast approach for positioning functions, favoring incremental development. The justification of the work is that the previous techniques are not suitable for use in the industry, because they are implemented as optimizations applied at the end of development, contradicting the inclination to the incremental techniques used in those institutions. They argue that the rationale for positioning functions rather than basic blocks is that compilers generally do not support block reordering. The idea behind the technique is that annotated call graphs may not be expressive enough to capture all possible sources of *cache* conflict. Effects such as subsequent calls to distinct functions within loops are not captured. To circumvent this problem, a special type of graph called the “loop call graph” is used, which concurrently models aspects of normal calls and nested loop calls. This technique is not based on WCET analysis, being applied generically throughout the program (all possible paths). This means that improvements can be observed both in relation to ACET and WCET.

4.3.4 Through *scratchpad* allocation

Caches are problematic for critical real-time systems due to the difficulty of (FALK; LOKUCIEJEWSKI, 2010) access prediction. WCET estimates, in some cases, can be considerably overestimated by the existence of *caches*. Optimization techniques that focus on *cache* effects to reduce WCET do not always solve the problem. So, what is done in practice in critical real-time systems is the disable of this type of resource, leading to a very low average case performance, since each access to the memory ends up being served directly by the main memory, which generally operates in speeds much slower than the processor. An alternative to *caches* are the scratchpad memories, which provide good performance for both the worst case and the average case. There are some techniques in the literature that promote the reduction of WCET by allocating data or executable code segments to the scratchpad memory whenever possible.

In (SUHENDRA et al., 2005) is presented a technique for static allocation of data in *scratchpad* memory. In this technique, the data allocation is performed using an combination of ILP with *branch-and-bound*, considering that all execution paths are feasible. It is considered static because the same data remains in the *scratchpad* during the entire execution of the program.

In (DEVERGE; PUAUT, 2007), the authors present a dynamic and hybrid strategy for allocating data in *scratchpad* memory. It is considered hybrid because it combines ILP with iterative heuristics. According to this technique, at each iteration, the WCEP is computed using a WCET analyzer, then it is decided which data will go to the scratchpad using ILP. It is dynamic because the data is placed and removed from scratchpad at run time, resulting in a software-controlled cache.

In (WAN et al., 2012) two techniques are proposed for dynamic allocation of data in *scratchpad* that offer better results than the technique of (DEVERGE; PUAUT, 2007). The first technique consists of a heuristic that selects variables to be stored in the scratchpad based on their impact on the longest k paths of the program. The second heuristic uses the problem of coloring graphs, analogous to the allocation of registers.

In (PUAUT, 2006) a technique is presented for selecting content in instruction caches with lockable content. Caches with lockable content behave similarly to scratchpad memories. This approach is based on the knowledge of the access pattern of a program, and does not necessarily provide optimal results.

In (FALK et al., 2007) they use an explicit search through the WCEP to select candidate blocks to lock in *cache*. Such a technique is highly costly since it consists of continuously exploiting the program's CFG. A similar technique is proposed in (PUAUT, 2006), where multiple optimization steps are applied considering a single WCEP path. After applying these steps, the WCEP is recomputed and the algorithm continue with the next steps.

In (FALK; KLEINSORGE, 2009) is presented a static and optimal technique that allocates executable code in scratchpad memory based on the work of (SUHENDRA et al., 2005). This technique is considered optimal because it generates the smallest possible WCET, considering the use of *scratchpad*. For each basic block, two WCET values are calculated: (1) considering the existence of a cache and (2) considering the execution of the entire program in a hypothetical scratchpad of arbitrary size. With WCET values and flow description, the decision on the destination of a basic block, whether in normal memory or *scratchpad*, is made using ILP. The latter technique can also be used for data allocation, according to (FALK; LOKUCIEJEWSKI, 2010).

4.4 CHAPTER SUMMARY

In this chapter we presented the most relevant optimizations in the literature for reducing WCET at compile time. The techniques considered include code transformation at program level, assembly level, source code level, data and instructions in scratchpad memory allocation, registers allocation and static branch prediction. Any technique aimed at reducing the WCET must be based on the use of the WCET analysis itself. The efficiency of the techniques is also related to the treatment given to the so-called WCEP switch effect. Techniques that recalculate WCET to verify WCEP switches tend to obtain better results. If the WCEP switch is not considered, then we run the risk of optimizing code that will not even contribute to the WCET of the application. In this chapter we have presented versions of classic optimizations aimed at reducing WCET, such as: loop unrolling and loop unswitching techniques.

The techniques presented are summarized in Table 3. In this table, the techniques appear in the order they were presented in this chapter. The columns indicate the name of the technique, followed by the type of strategy adopted. It can be seen in this table that some techniques rely on the transformation of source code while others in the assembly representation of the programs. Optimizations can also work on the program's layout, that is, the way that parts of the program are mapped into memory. Finally, some techniques move parts of the program or data into the scratchpad memory, especially those that directly impact on the application's WCEP/WCET. This use of *scratchpad* may be static, where the desired content is loaded before program execution, and remains in this memory until the end of it. It can also be dynamic, where the content that will go to *scratchpad* is defined *a priori* (at compile time), but copying data from memory to it, or vice versa, is done by the software. In this latter strategy, the use of *scratchpad* behaves like a *cache* with software-controlled substitution policy. We will retake an overview of some of the related works to clarify some specific issues in the chapters related to the contributions.

	Source code opt.	**Assembly opt.**	**Layout opt.**	**Scratchpad**
WCET-aware procedure cloning (LOKUCIEJEWSKI; FALK, 2008)	X			
Superblock optimizations (1) (*WCET-aware superblock scheduling*) (LOKUCIEJEWSKI et al., 2010)	X			
WCET-aware Loop Unrolling (LOKUCIEJEWSKI; MARWEDEL, 2010)	X			
WCET-Aware Loop Unswitching (LOKUCIEJEWSKI; MARWEDEL, 2009)	X			
WCET-aware Function Inlining (LOKUCIEJEWSKI; MARWEDEL, 2011)	X			
Superblock Formation (ZHAO et al., 2006)		X		
WC Path Duplication (ZHAO et al., 2006)		X		
Loop Unrolling (2) (ZHAO et al., 2006)		X		
WCET-aware trace scheduling (LOKUCIEJEWSKI; MARWEDEL, 2011)		X		
Static branch prediction for WCET reduction(HUANG et al., 2012).		X		
Register allocation by graph coloring considering WCET (FALK, 2009)		X		
Register allocation by ILP considering WCET(FALK et al., 2011)		X		
Register allocation together with instruction rescheduling considering WCET (HUANG et al., 2012).		X		
Basic block positioning to improve branch performance (ZHAO et al., 2005)			X	
Function positioning to improve *cache* (LOKUCIEJEWSKI et al., 2008)			X	
Function positioning by COP (MARREF; BETTS, 2011)			X	
Incremental function positioning (MEZZETTI; VARDANEGA, 2013)			X	
Static allocation of data in scratchpad with ILP + *branch-and-bound* (SUHENDRA et al., 2005)				X
Dynamic data allocation in *scratchpad* with with ILP + heuristic search (DEVERGE; PUAUT, 2007)				X
Dynamic data allocation in *scratchpad* by longest paths and graph coloring(WAN et al., 2012)				X
Content selection by access pattern for instruction caches with lockable content (PUAUT, 2006)				X
Content selection by access pattern for instruction caches with lockable content (FALK et al., 2007) using an explicit search through the WCEP *cache*.				X
Selection of blocks of instructions for scratchpad with optimal result(FALK; KLEINSORGE, 2009)				X

Table 3 – Summary of techniques for WCET reduction

5 EXPERIMENTATION INFRASTRUCTURE

This chapter provides an overview of the experimentation infrastructure used for the development and validation of the thesis. This infrastructure consists of a target processor, a compiler backend and a WCET analyzer. The compiler was based on an existing solution, while the WCET analyzer was built in a joint effort with other doctoral (STARKE, 2016) thesis. The analyzer has been developed using classical techniques for of pipeline analysis, cache analysis and search of the worst-case path. In the next section, we will describe aspects related to the target architecture.

5.1 ARCHITECTURE AND REFERENCE PROCESSOR

The architecture used as the target of this work was developed using VHDL in another doctoral work (STARKE, 2016) (STARKE et al., 2017). Such architecture is based on the ST231 (STMICROELECTRONICS, 2004) processor, which is a member of the ST200 series. Some relevant features inherited from ST231:

- Parallel execution units, including ALUs (arithmetic logic units) and multipliers.
- Predicated execution through *select* operations.
- Efficient branch architecture, with condition registers or *flags*.
- Immediate encoding greater than 32 bits.

The ST231 is a very long instruction word (VLIW) processor. VLIW processors use a technique in which instruction-level parallelism is explicitly exposed by the compiler. In VLIW processors, a set of RISC-style operations are grouped into packages or bundles. All operations present on a given bundle are executed in parallel. While the delay between issuing a instruction and its ending is the same for all instructions, some internal results may be available early through bypassing paths.

VLIW processors are simpler than superscalar processors. This is due in part to the fact that instruction scheduling is performed by software development tools that are used in support for architectures. Superscalar processors have dedicated hardware logic for dispatching concurrent and/or out-of-order operations.

5.1.1 Registers

The processor features a bank of 64 32-bit registers, where the only non-general purpose registers are the LR (*link register*) with number 63, which is changed by procedure call statements and the ZERO register, which always retain the value 0. There are still some general purpose registers which are conventionally used for specific purposes. Such registers are:

SP: Stack pointer, whose number is 12.

GP: Global pointer used for reference to certain data types, whose number is 13.

TP: Thread pointer, whose number 14.

The other registers should be used according to the procedure call convention, which will be detailed later.

Another bank with 8 1-bit registers is still available. These registers, called branch registers, are used as flags for branching operations and to store *carry* values of some arithmetic operations.

5.1.2 Instructions

Basically, the processor implements the same general instructions of the ST231. Except for division operations support. The ST231 features only one DIVS (non-restoring division stage) instruction that allows the implementation of software divisions, while the processor used allows hardware divisions. Like ST231, there is no support for floating-point operations. The conditional branch instructions are predicated, that is, they depend on previously calculated values, which can be true or false.

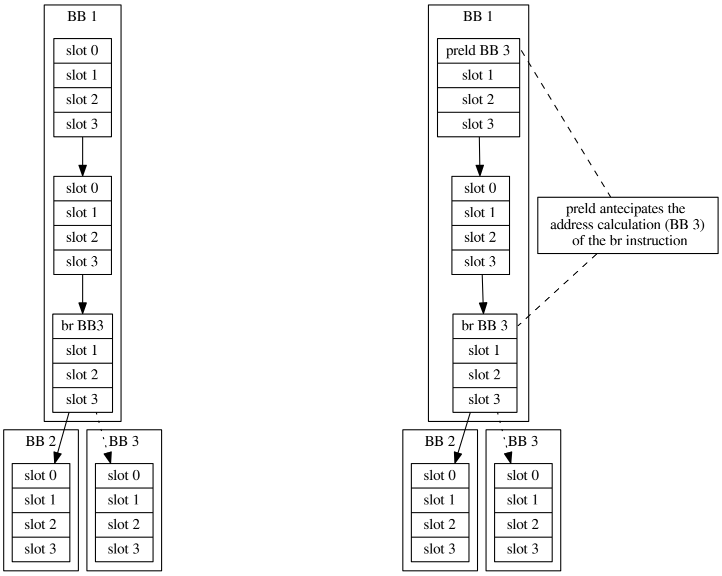
5.1.2.1 Branch prediction instructions

Another difference in relation to the ST231 is the static branch prediction support. The processor used has a new special instruction that allows the compiler to indicate whether a given path is more likely to be executed. That instruction is called *branch preload* or *preld* at ISA level (STARKE et al., 2016). When a branch is more likely to be taken, a *preld* instruction can be scheduled in a previous bundle. The overhead of the *preld* instruction/operation will be zero if we can find a free slot in an existing bundle. The exact position of a *preld* operation is 2 cycles before a branch operation. The *preld* instruction works by anticipating the calculation of the branch target address, forcing the branch ahead to follow a *taken* behavior. In this way, the *preld* instruction emulates the existence of an entry in a branch target buffer (BTB) relative to the next branch. If such a *preld* instruction does not exist for a determined branch, it has a *not taken* or *fall-through* behavior.

A usage example of the *preld* instruction is presented in Figure 17. In Subfigure 17a, we want a *not taken* behavior for the branch operation in the third bundle of basic block 1, so we do not need a preload instruction. Alternatively, if we want a *taken* behavior for the same branch, we must insert a *preld* instruction 2 cycles before the referred branch, as showed in Subfigure 17b.

5.1.2.2 Predicated execution of instructions

The used processor has an ISA extension that enables code predication in a simplified way through the thirtieth bit, which is otherwise unused. If an operation has its 30th bit enabled, then the result of the instruction will only be committed if the predication flag is configured to true. If the predication flag has false as value, then the operation will produce no effect (or nop). The predication flag is a 1-bit register



(a) Without branch preload (not taken behavior)

(b) With branch preload (taken behavior)

Figure 17 – Example of a sequence of basic blocks with and without branch preload. Solid lines mean adjacent basic blocks/bundles, whereas dotted lines denote successor basic blocks that are reached by branch instructions. With the preld instruction the address calculation of the branch instruction is anticipated to reduce penalties in the case of a correct prediction

that can be accessed through comparison instructions. This flag is connected to the branch register number 4 that is already defined by the ISA. In this way, the branch register number 4 controls the execution of predicated instructions. To enable or disable the predicated execution mode of the processor, two instructions where added: *par_on* to enable and *par_off* to disable.

Table 16 and 17 of Annex A, shows operands used in instructions and all instructions of the used architecture, respectively.

5.1.3 Processor organization

The processor is able to start the execution of 4 operations at each clock cycle. This means that it has 4 parallel pipelines. Although they have the same number of stages, each pipeline has access to different functional units. The functional units available in each pipeline are shown in Table 4. Each *pipeline* provides in-order execution and presents 5 stages.

Table 4 – Execution units in each pipeline

Pipeline 0:	Call/Branch	Mult	Div	Alu	Mem
Pipeline 1:	Mult	Alu	Mem		
Pipeline 2:	Alu				
Pipeline 3:	Alu				

Graphically, we can see the simplified data path in Figure 18.

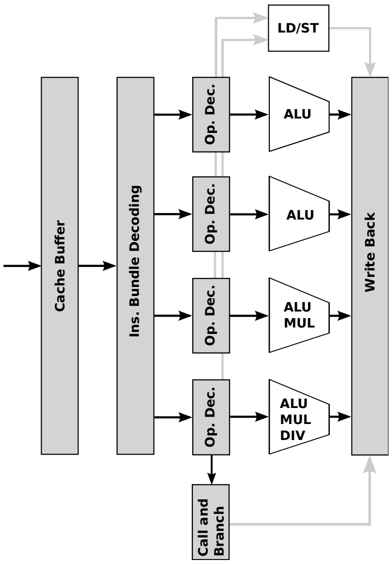


Figure 18 – Data path of the processor

Regarding the memory hierarchy, the processor has a direct mapped instruction cache, where each line contains 8 32-bit words, with a total of 32 rows. There is no data cache, but there is a configurable *scratchpad* memory.

Regarding forward paths, the following connections exist on the processor:

- Output of the ALUs for the input of the ALUs.
- Output of the ALUs to the memory access unit.
- Output of the ALUs to the call/branch unit.

However, for the following situations there is no forward path, but there is interlocking:

- Output of the ALU to the input of the conditional branch unit. If a branch operation (in a bundle) requires the value of a register recorded in an operation of the previous bundle, the interlock will put a stall between the two bundles.

5.1.4 Instruction coding

As we are dealing with a VLIW architecture, sets of operations must be encoded in a single instruction. Each instruction can have a variable number of 1 to 4 operations. The configuration presented in Table 4 imposes the following restrictions on the organization of valid *bundles*:

- Because the processor has only one call/branch unit, only one of such operation can be included in a bundle. If such an operation exists, it must be the first one in the coding.
- There are only two multiplication units. This means that at most 2 multiplication operations can be encoded in a bundle. Such operations shall be coded as the first and/or second operation of the bundle.

- Only one division unit is available. Divisions should be encoded as the first operation of a bundle.

- Logical and arithmetic operations can reside in any position of the bundle, since all pipelines have an ALU.

In addition to the above restrictions, there is still one more related to memory operations. Although both pipeline 0 and pipeline 1 can accommodate memory access operations, only one can be contained in a bundle.

The coding adopts a compression strategy. Instead of inserting empty spaces in bundles to represent *nop* instructions (empty operation), we compress bundles using *stop bits*. When an operation must be the last one in a *bundle*, it must have its most significant bit set to 1.

Like other VLIW architectures, there are code alignment constraints related to *cache*. A *bundle* must be entirely contained in a *cache* line. This means that the end of a *cache* line should always match a *bundle* end (*stop bit* enabled). Alignment is done by inserting *nops*.

5.1.5 Procedure calling conventions

The procedure calling convention used is not the ST231 standard. A specific convention, defined as follows:

Registers to argument passing: Arguments must be passed by the following registers, which can also be used to return values: R16, R17, R18, R19, R20, R21, R22, R23.

Preserved registers: The registers that must be preserved are: R1, R2, R3, R4, R5, R6, R7.

Scratch registers: Registers that do not need to be preserved by called functions are: R8, R9, R10, R11, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62.

Actually, the registers TP and GP are not used.

5.2 CODE GENERATION FOR THE REFERENCE ARCHITECTURE

A new code generator was developed in LLVM as compiler infrastructure. LLVM (LATTNER; ADVE, 2004) is a modern and extensible infrastructure for compilation and optimization. It is composed of a large number of tools, ranging from profilers/interpreters and even just-in-time compilation helpers. In its usual manner, a LLVM-based tool chain resembles the diagram of Figure 19. In this scheme, one can observe that the compilation process is shared by both Clang [1], which is the front-end of the compiler, and LLVM. LLVM is also responsible for assembling the code, i.e., object code generation. The finalization, which is the linking of the compiled code, must be performed by an external tool, such as with *ld* present in the package *binutils* [2].

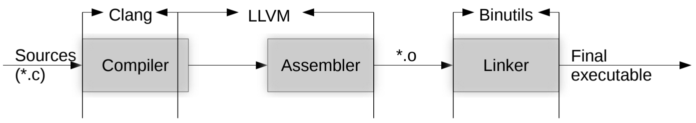


Figure 19 – Possible use of Clang with LLVM and linking with Binutils

LLVM is fundamentally organized in passes, and everything is implemented as passes. The fundamental unit for representing programs in LLVM is the *Module* class, which is an intermediate code unit containing as data global variables, symbols and function instructions. Passes can operate over modules, functions, loops, regions, and other units that can compose a program. The passes are managed and scheduled by what is called *Pass Manager*. A pass can analyze or modify the

[1] http://clang.llvm.org
[2] http://www.gnu.org/software/binutils

intermediate representation of the program (optimization), and may be dependent on information/analysis generated by other passes. To ensure that the information is available when needed, the pass manager schedules passes in order to solve all required dependencies.

When we have a module containing functions and a set of passes to execute through this module, all these passes are executed on each of these functions in a chained way that are isolated from other functions. We call this chained execution as pipeline. This strategy of pipelined execution of passes ensures efficient memory and cache utilization, because analyzes are performed in very close time intervals relating to their use (temporal locality). When all passes are executed on a function, the pass manager can safely deallocate all data relating to the analysis of this function, since they are no longer needed for the next function. If code generation must occur, it will be performed by the last passes executed on a function. It can be said that the LLVM is stateless with respect to analysis data, when considering isolated functions.

However, as mentioned above, LLVM only comprises the backend functions of the compiler, requiring a frontend to complete the infrastructure. An existing frontend that operates in conjunction with the LLVM is Clang, which supports almost the entire family of languages derived from C (C, C ++ and Objective-C). We did not need to change Clang because our work addresses code generation and not programming languages. We only added information needed by Clang to recognize the new target and use the appropriate tools that were developed for the architecture, such as the assembler and linker, for example.

5.2.1 Back-end support

The compilation infrastructure can be outlined by Figure 20. Note the use of a customized linker instead of an existing solution.

The implementation of the support for the new architecture was performed as follows:

- The register of a new compilation target (LLVM *Target*). This

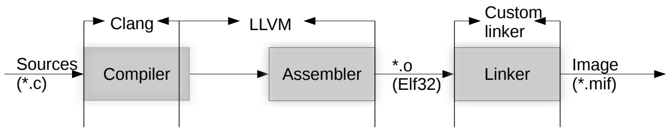


Figure 20 – Use of Clang with LLVM and finalization with a custom linker

is accomplished by adding a set of classes that extend abstract classes previously existing in the compiler, as it is developed according to the paradigm of object orientation.

- Specification of instructions and registers, as well as their coding. This is done through a tool called *TableGen*. This tool helps us keep records of information on specific areas of the compiler, thus avoiding duplication of information. Specification using TableGen enables automated generation of parts of the backend as assembly generator, instruction selector and instruction encoder.

- Implementation of passes responsible for lowering (or legalization[3]) IR (intermediate representation). Code legalization includes:

Type promotion: Types of reduced representation (one bit for example) that are not natively supported should be promoted to larger representation types (8, 16 or 32 bits for example).

Type expansion: A type that requires more bits than the larger word supported by the processor, should be expanded in several of these words.

Operation expansion: Intermediate representation operations not natively supported by the processor should be expanded in a

[3] It is called legalization because common operations from the intermediate representation of the LLVM are converted into operations that are legally supported by the target architecture. An operation is considered illegal if it has no equivalent in the target architecture

series supported operations. For some operations, the compiler does not require the backend for such a transformation. Though, some complicated transformations must be done by the backend.

We also use the standard VLIW instructions scheduler of LLVM. The scheduler takes into account the structural dependencies specified in the description of the instructions, via definition of scheduling itineraries for each of these. Such descriptions are used in a class called *Hazard Recognizer*, which provides access to resource reservations tables, necessary for scheduling. This is used in conjunction with the instruction priority DAG. The instruction scheduling is performed in a stage prior to register allocation, then the instructions are still represented in SSA (*static single assignment form*) form.

Two another passes are still necessary to complete the code generation process:

Instruction packetization: The instruction scheduling, which executes on stage prior to register allocation, only reorders individual instructions. This pass implements packetization routines used to define which instructions must form each bundle after register allocation. In assembly, the semicolon is used as end of bundle marker. Given the sequences of instructions generated by the scheduler, the packaging should take into account the following restrictions:

Data dependency: In a same bundle, instruction should not have certain types of data/registers dependencies. Such dependencies can be:

Data: This kind of dependence is also called true dependence, where an instruction defines the content a register that will be used by another instruction. Instructions with data dependencies must reside in distinct bundles, respecting the order of execution. An example of this type of dependence can be seen in the code 5.1.

In this example there is a data dependency between the second and third operations, in relation to the register $r18$.

Code 5.1 – Example of data dependency between instructions

```
add     $r11 = $r19 , $r10
shru    $r18 = $r18 , 11
add     $r9 = $r9 , 1
        ;

sth     2[$r10] = $r18
        ;
```

Output: In this type of dependence, two instructions write to the same register. As in the previous case, instructions with output dependency must be placed in different bundles, respecting the order defined in the scheduling.

There is still a kind of dependence called anti-dependency. In this type of dependency, an instruction writes a register previously read by another instruction. This kind of dependency is ignored in the packaging phase, because it does not represents a problem for the used architecture. An example of packaging considering and disregarding anti-dependencies can be seen in Figure 21.

Multiple control flow instructions: Each bundle must contain at most one control flow instruction, as conditional/unconditional branches and procedure calls.

Multiple memory access instructions: Each bundle must contain at most two memory access instructions, whether they are loads or stores.

Long instructions: Some instructions can use two entries in a bundle. As example we can cite instructions with 32-bit

```
sth     0[$r10] = $r19
add     $r16 = $r16, 2
        ;

shr     $r10, $r18, 18
add     $r8 = $r8, 1
        ;
```

```
sth     0[$r10] = $r19
add     $r16 = $r16, 2
shr     $r10, $r18, 18
add     $r8 = $r8, 1
        ;
```

Figure 21 – Packaging instructions considering anti dependencies (left), and disregarding (right). Note that there is a anti-dependency in relation to the register $r10$ between the first and third instructions.

immediate. This type of instruction can not be divided in two bundles.

Bundle alignment: The architecture requires that the bundles are aligned in cache. When a cache line is fetched from memory, it must contain only complete bundles, i.e., there will be a stop bit in the last instruction from this line. Without an additional alignment pass, we can have bundles that pass the cache line frontier, which is an illegal situation from the processor point of view. The alignment procedure operates by shifting code using nop instructions.

The alignment procedure operates in one function at a time. For this purpose, it considers that function beginnings (first bundle) are aligned, which should be respected by the linker later. Starting from the first bundle, the subsequent bundles are checked. At this point, two situations may occur:

- Aligned bundle: nothing must be done, because the bundle is already aligned.
- Unaligned bundle: the unalignment must be corrected. For this purpose, we calculate how many instructions are necessary to put in this line to shift the unaligned bundle entirely to the next cache line. The calculated instructions are then

translated into nop instructions that are placed in the previous bundle to make the shifting of the current bundle.

At the end, more nop operations are added to the end of the function, so that the next function will start at an aligned memory position, as required by the algorithm. Figure 22 shows an example of bundle alignment. In this example we consider that the first bundle starts in a aligned memory position (start of a cache line) and each cache line holds up four words. So, the third bundle is unaligned (divided in two cache lines). To fix this unalignment, we put a nop operation in the second bundle.

```
shl     $r16 ,  $r8 ,  2
        ;

add     $r17  =  $r16 ,  $r9
add     $r8  =  $r8 ,  1
        ;

// unaligned bundle
ldw     $r17  =  0[$r17]
add     $r16  =  $r16 ,  $r10
        ;

stw     0[$r16]  =  $r17
        ;
```

```
shl     $r16 ,  $r8 ,  2
        ;

add     $r17  =  $r16 ,  $r9
add     $r8  =  $r8 ,  1
// alignment nop
nop
        ;

ldw     $r17  =  0[$r17]
add     $r16  =  $r16 ,  $r10
        ;

stw     0[$r16]  =  $r17
nop
        ;
```

Figure 22 – Alignment instructions in a basic block. Considering that the initial address of the basic block is aligned with the beginning of a cache line, then in the example on the left, the third bundle is misaligned. In the example on the right example, the basic block was aligned by inserting nops.

Promotion memory references to registers: One characteristic of the implemented architecture is that it does not have data cache, due to the difficulty of analyzing this type of resource considering WCET. So, any memory access will result in large amount of processor stalls. One of the major sources of memory access identified in the compiler comes from the use of the procedure stack in which each access is usually preceded by a load operation and followed by a store operation. In this way, we can try to promote as much data as possible from stack to register, as the architecture has a large number of them. By default, this is not done by the LLVM, however there is an available pass that performs this procedure at intermediate representation level, i.e., before the code generation. The pass in question is based on the work of (SREEDHAR; GAO, 1995).

Determination of loop bounds: We also implemented a pass to discover worst-case iteration counts of all loops. This execution counts are necessary to WCET analysis. The compiler itself can estimate worst-case counts for simple and data-independent loops. For more complicated or data-dependent loop structures, those worst-case execution counts must be provided through annotations in the source code. Figure 5.2 shows an example of loop annotation. In this example, the inner loop is data-dependent on the outer loop, with a worst-case bound of 5 iterations.

Code 5.2 – Example of data dependency between instructions

```
for(i = 0; i < 5; i++){
   j++;
   //@loop-bound: 4
   for(int m = 0; m < i; m++){
     j++;
   }
}
```

If the compiler cannot bind an iteration bound for a loop, the compilation is aborted and a message is shown to inform which loop must be annotated by the user. These loop bounds are exported within the CFG, as we will show soon.

Static branch prediction (optional): An optional pass was implemented to perform static branch prediction. LLVM perform analyses that can be used to statically predict the behavior of a branch. Such analyses can, for example, estimate the probability of each branch to be taken by the program. These probabilities are obtained through a set of heuristics that act on the structure of loops and its exit conditions. The examination of comparison operations that precede a branch instruction is also a key aspect of these heuristics. In this case, we used branch probabilities obtained from the *Machine Branch Probability Info* pass, which is an analysis pass available on the LLVM infrastructure. This strategy is summarized by Algorithm 2.

Algorithm 2 Algorithm for static branch prediction using compiler information.

```
1: procedure SET_PREDICTIONS($CFG$) ▷ Set predictions to each branch
2:     $branch\_probability\_analyzis(CFG)$
3:     for all $BB \in CFG$ do
4:         if $is\_conditional\_branch(BB)$ then
5:             if $prob(BB, BB.tk) > prob(BB, BB.ft)$ then
6:                 $BB.direction \leftarrow taken$
7:             else
8:                 $BB.direction \leftarrow fall-through$
9:             end if
10:        end if
11:    end for
12: end procedure
```

When a branch is predicted as *taken*, we insert a *branch preload* instruction 2 cycles before such operation, according to the static branch prediction support of the architecture.

CFG extraction: An additional step is still executed to extract the control flow graph of the application. This graph is useful for WCET analysis, as will be seen later. Each graph node stores information relevant to the basic block represented. If a node represents a loop header, then we also associate it with worst-case iteration counts previously obtained. We also export information related to control flow operations. By default, all branches of a program are statically predicted as *not taken*, but these predictions can change if we enable the *static branch prediction pass* previously explained. In these way, considering a basic block terminating with a branch/jump/call/return, we can have 7 types of successors that are exported in the CFG:

Jump successor: The successor is reached through a jump operation.

Call successor: The successor is reached through a call operation.

Return successor: The successor is reached through a return operation.

Not taken successor: This case represents that the successor basic block is the not taken target of the actual block, considering a default *not taken* prediction.

Taken successor: This case represents that the successor basic block is the taken target of the actual block, considering a default *not taken* prediction.

Not taken successor (predicted as taken): This case represents that the successor basic block is the not taken target of the actual block, considering a *taken* prediction.

Taken successor (predicted as taken): Finally, this case represents successors that are in the taken target of basic blocks, considering a *taken* prediction.

In the current version, a single graph is generated for the entire program. In this graph, each procedure call is replaced by a full copy of the called procedure. Then, it is considered that this graph is a composition of the control flow graph with the graph of procedure calls, as a single graph stores all the information. As an example, we can consider the Code 5.3, which represents a C program that calculates fibonacci numbers in an iterative way. The procedures CFGs *fib* and *main*, can be seen in Figures

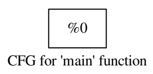
23b and 23a, respectively. The composition of these two graphs results in the graph that is shown in Figure 23c, where the numbering represents the unique identifier of each node. This graph is called Interprocedural Control Flow Graph (ICFG) (WILLIAM; BARBARA, 1991).

Code 5.3 – Program that calculates fibonacci numbers in an iterative way

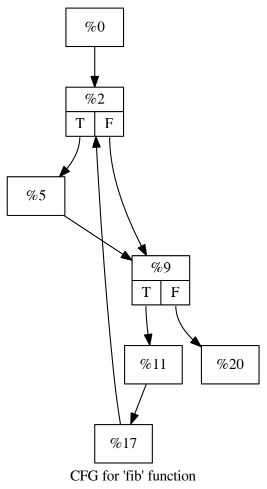
```
int fib(int n){
  int  i, Fnew, Fold, temp,ans;

    Fnew = 1;  Fold = 0;
    for ( i = 2;
          i <= 30 && i <= n;
          i++ )
    {
      temp = Fnew;
      Fnew = Fnew + Fold;
      Fold = temp;
    }
    ans = Fnew;
  return ans;
}

int main(){
  int a;

  a = 30;
  fib(a);
  return a;
}
```

5.2.2 Code Linking

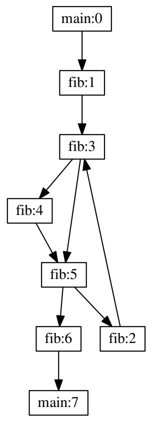
The code generated by the compiler can not be directly executed. It is still necessary to define the program layout, ie, how it should be mapped into memory for execution on the target processor. Compilers generally do not make any addresses definition, they only provide



(a) CFG of the main function



(b) CFG of the fib function



(c) Complete CFG of the fibcall program

Figure 23 – Example illustrating the formation of a complete CFG (or ICFG)

some relocation tables with the binary, which describe what should be adjusted and configured in the program. When the address for an item of data is defined, we should adjust each instruction that accesses this data, so that it can use its correct address. Entries or records in the relocation table store the position of instructions and the type of adjustment that must be made to access a specific data item. All this information is in the object code, which is stored in a file called ELF (*Executable and Linkable Format*) (SCO, 2013). In this file type, text sections, data and constants, symbol tables and relocation tables are stored in specific sections.

In addition to the definition of program layout and relocation of symbols, it is also the linker's task to define what will be the final format of the file that contains the executable code. As we are using an architecture implemented in FPGA, our linker generates a MIF file (*Memory Initialization Format*) used to initialize the ROM memory. In this case, none initialization of data is performed by the linker, the application itself must copy and initialize the data from ROM to RAM. So, for execution purposes, every application is compiled with initialization routines, that is, a boot loader.

5.3 WCET ANALYSIS

The implementation of a WCET analyzer considered the choice of the most appropriate and accurate techniques, but kept the basic premise of simplicity of analysis. The chosen techniques basically regards cache analysis, pipeline analysis and finally worst-case path search.

The input for the WCET analyzer is a compiled and link-edited program, and a CFG. The CFG is obtained from the compiler, as described above. By using the CFG obtained from the compiler we have some advantages:

- We do not need to reconstruct the CFG directly from the machine code, because this information is already available in the compiler;

- Recognition of structures that are hard to be reconstructed by static analysis, as jump tables (indirect addressing) used to implement switches. The compiler always knows all possible targets of a switch statement.

The compiler also must generate code in a pattern that allows complete construction of the CFG. This pattern includes absence of indirect function calls and indirect recursion. These assumptions are not too restrictive and are adopted by many WCET analyzers. The steps executed by the analyzer can be summarized as follows:

Loop detection: loops are not described in the CFG. The detection of loops uses the Tarjan's algorithm (LENGAUER; TARJAN, 1979) for identifying strongly connected components in graphs.

Instruction cache analysis: analysis for classification of cache accesses in *always miss*, *always hit*, *first miss* and *conflict*.

Detection of memory accesses: Detection of accesses targeted to the main memory and not to the scratchpad memory. This analysis is quite simple and work by inspecting load and store instructions, so it will not be detailed.

Stack utilization analysis: This analysis calculates the stack utilization of a program by interpretation of function calls. This analysis is useful to guarantee that the target program will respect its stack budged. This analysis does not affect the resultant WCET, but helps to avoid possible stack overflow errors.

Pipeline analysis: This analysis calculates the basic block times when executed in the processor pipeline, disregarding cache effects.

Worst-case path search: This phase searches for the worst-case path and its respective computation time (WCET), considering the previous analyzes. We used IPET (LI; MALIK, 1995) for this purpose, as we will show further.

Since the used architecture does not suffer from timing anomalies, we can conduct each analysis isolated and combine the results at the path search phase.

For the remainder of this section, we will consider the example of Figure 24, consisting of a C program and its corresponding CFG. In this example there is a program consisting of a loop and an if-then-else sentence. In the CFG of the example, the start node is explicitly drawn and is purely symbolic, not representing any real basic block. There is another symbolic node representing the end of a program in the CFG, but this node is omitted from the drawing. The edges follow the definition presented earlier, where an edge $di\_j \in E$ means the basic block j can be executed after the execution of the basic block i.

```
int main(int argc, char** argv){

  int i = 1;
  int j = 0;

  for(j = 0; j < 5; j++){

    if(i < 6){
      i++;
      i+=1;
      i+=2;
      i+=3;
    } else {
      i--;
      i-=1;
      i-=2;
      i-=3;
    }
  }
}
```

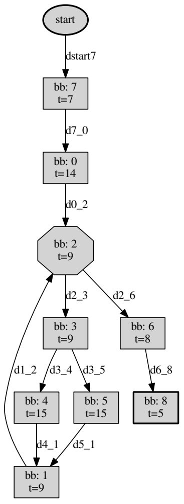


Figure 24 – A C program and its respective CFG

Like most WCET analyzers, the implemented one is context sensitive. The analyzer considers the paths by which each node of the CFG can be reached, and computes the behavior of *cache* and executions for each node in each of these contexts. For example, loops may have a different execution time for the first iteration, when instructions must be loaded to the *cache*. This behavior can be extracted by cache analysis, which is the subject of the next subsection.

5.3.1 Instruction cache analysis

Cache memories are necessary to minimize the gap between processor and memory performances. Usually, main memory has a clock that is slower than the processor so a fast cache is placed between them where the most recent data is stored promoting faster access. We use a direct-mapped instruction cache memory, so some sort of analysis is necessary to model cache misses during WCET analysis. Usually cache analysis is performed using abstract interpretation as described in (ALT et al., 1996) but here we used traditional data flow analysis to compute cache states. Similar analyzes are also used in (MUELLER; WHALLEY, 1995) and (LEE et al., 1998).

A cache memory is characterized by its capacity, line size and associativity.

Definition 8. *(Capacity) Capacity is the caches' total number of bytes.*

Definition 9. *(Line or block size) Line size is the quantity of bytes transferred from memory to cache when a cache miss occurs. A cache could have $n = \frac{capacity}{line.\ size}$ lines.*

Definition 10. *(Associativity) It consists of the mapping of various main memory addresses to cache lines. A direct-mapped cache has unitary associativity, where a main memory specific address is always mapped to the same cache line. If associativity is 2, a main memory address is mapped to 2 different cache lines. When associativity is not unitary, some sort of replacement policy must exist to decide which line will have data for eviction. The relation $\frac{n}{assoc.}$ defines the number of sets of a cache.*

In the context of this work, we will focus only on the unitary associativity (direct mapping) where one memory block can reside only in a specific cache line. This feature does not impose a restriction upon the analysis, but is related to the used direct mapping configuration in the hardware. A direct-mapped cache memory is formed by a sequence of lines $L = l_1, l_2, l_n...$ which store a set of memory blocks $M = m_1, m_2...m_s$. A memory block m with address $addr$ is stored in the line l_i following Equation 5.1. The operator % represents the modulus or remainder of the division.

$$l_i = addr(m) \ \% \ n \tag{5.1}$$

In the case of the existence of a particular instruction in the cache, we define:

Definition 11. *An instruction could be in the cache if: 1) there was a CFG transition sequence where the block corresponding to the instruction memory had been referenced in previous basic blocks; 2) this memory block is referenced previously in the same basic block.*

Definition 12. *(Abstract state) An abstract state of the cache of a basic block is the subset of all memory blocks that can be cached before executing the basic block.*

Definition 13. *(Reachable abstract state) A reachable abstract state is the subset of all memory blocks that can be reached by CFG transitions.*

Definition 14. *(Effective abstract state) An effective abstract state of a basic block is a subset of all memory blocks that can be reached by considering all the CFG paths to the basic block in the analysis.*

5.3.1.1 Reachable and effective abstract state

We can construct reachable abstract state where we can map all memory blocks accessed by every basic block. This analysis uses the same principles of reaching definitions in traditional data flow analysis and it follows Algorithm 3.

Figure 25 shows an example of the results of Algorithm 3 execution. The abstract reachable state ($RMB_{bb}(cl) = data$) is beside each

Algorithm 3 Reachable abstract state for every basic block

```
 1: $change \leftarrow true$
 2: while $change$ do
 3:     $change \leftarrow false$
 4:     for all $i \in BB$ do
 5:         for all $c \in cache\_blocks(i)$ do
 6:             ▷ $p$: predecessors of $i$
 7:             $\mathrm{RMBin}_i(c) \leftarrow \bigcup_{\forall p} \mathrm{RMBout}_p(c)$
 8:             $temp \leftarrow RMBout_i(c)$
 9:             ▷ If this cache block is the last accessed by $i$
10:             if $last_i(c) \neq \emptyset$ then
11:                 $\mathrm{RMB}_i(c) \leftarrow last_i(c)$
12:             else
13:                 $\mathrm{RMB}_i(c) \leftarrow \mathrm{RMBin}_i(c)$
14:             end if
15:             if $\mathrm{RMB}_i(c) \neq temp$ then
16:                 $change \leftarrow true$
17:             end if
18:         end for
19:     end for
20: end while
```

output edge. bb is the basic block number, cl is the cache line and $data$ represents an identifier of the memory data, the memory address index. Figure 25 also shows which cache line is accessed by a basic block and its memory address index inside the nodes ($l_0 = 0$ for basic block 0 and $l1 = 1$ for basic block 4). It is very easy to know the cache contents after execution of a basic block, we have only to inspect basic block instructions addresses. The RMBs track the possible state of cache lines after the "execution" of all basic blocks. We know, for instance, when bb_3 executes, cache line 0 should have data which index is 0 ($RMB_3(0) = 0$) because to reach basic block 3, bb_0 must execute. This same logic follows for $RMB_3(1) = 1$ and $RMB_3(2) = 0$, where $RMB_3(1) = 1$ comes from bb_2 and $RMB_3(2) = 0$ comes from execution of bb_2 itself.

Some conditions must hold when using this analysis to address cache hits and misses. First, loops must iterate at least once. If this is not true, we cannot assume a hit in bb_4 in Figure 25 because bb_2 will never execute and l_1 will never receive data index 1 used by bb_4. Secondly, some path checking should be done during the analysis. A hit could only exist in bb_4 if the worst-case path passes at least once

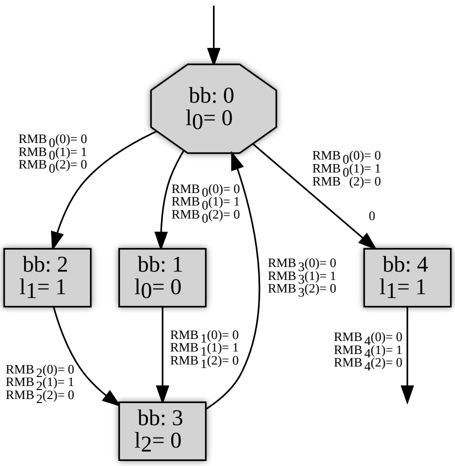


Figure 25 – Cache abstract reachable state example

through bb_2. In this case we are pessimistic and assume a miss in bb_4. This type of cache analysis could be optimistic if we use only abstract reachable states without path checking.

The path checking is performed constructing the effective abstract state using Algorithm 4.

Algorithm 4 Effective cache state for every basic block

```
 1: for all $i \in BB$ do
 2:     for all $c \in RMB_i(c) \vee |RMB_i(c)| = 1$ do
 3:         ▷ If this block is accessed by $BB_i$
 4:         if $last_i(c) \neq \emptyset$ then
 5:             $\mathrm{EMB}_i(c) \leftarrow c$
 6:         else
 7:             ▷ $P$: paths that leads to $i$
 8:             if $c \in last_j(c) \mid j \in \forall P$ then
 9:                 $\mathrm{EMB}_i(c) \leftarrow c$
10:             end if
11:         end if
12:     end for
13: end for
```

The effective abstract cache set $EMB_i(c)$ is constructed from reachable abstract set $RMB_i(c)$. First, on Line 2 it is checked if the set cardinality is 1 ($|RMB_i(c)| = 1$). If this cache block is accessed by the basic block, it is added in the abstract set on Line 5. Otherwise it checks if this memory reference is accessed by all paths leading up to the basic block in question.

Considering the example in Figure 25 and the basic block 4, the algorithm has the following execution in the construction of effective abstract state for the cache line 1 of the basic block 0 ($EMB_0(1)$):

- if vertex 0 accessed $l_1 = 1$, we could terminate and it will be classified as a cache hit;
- following vertex 0, we check 3, where there is no access;
- following vertex 3, we check 2. In this vertex, $l_1 = 1$ is accessed and this path search is ended;
- following vertex 3, we check 1. There is not a $l_1 = 1$ access and we continue;
- following vertex 1, we check 0. Vertex 0 was already visited. We can conclude that there is a path where $l_1 = 1$ is not referenced;
- $l_1 = 1$ does not belong to effective abstract state of basic block 0 and therefore there is a cache miss in basic block 4.

5.3.1.2 Cache accesses classification

We classify all program instructions in "always miss", "always hit", "first miss" and "conflict" after the data flow analysis in conjunction with path checking – reachable and effective abstract state.

Definition 15. *(A_MISS) There is a fault in the cache every time this instruction is executed – always miss.*

A_MISS classification occurs in compulsory or capacity faults. For example, a compulsory miss occurs in bb_0 in Figure 25 since the first instruction of the program is not in the cache memory at the beginning of the program execution. *A_Miss* is also the correct classification for the instruction of bb_4, since $l_1 = 1$ does not exist in the effective abstract state of bb_0 (predecessor of bb_4).

Definition 16. *(A_HIT) There is a hit in the cache every time this instruction is executed – always hit.*

A_HIT is the class for instructions where: 1) they are not the basic block first instruction or the first instruction of a cache line; 2) they are in the cache effective state. In the case of the Figure 25, all instructions of the bb_1 can be classified as hits, because its line will always be previously accessed by the bb_0.

Definition 17. *(F_MISS) This classification is related to loops. There is a cache miss only the first iteration of the loop – first miss;*

In the case of F_MISS, if a loop iterates 100 times, there is a cache miss only the first iteration. Others 99 iterations, there are cache hits. This classification occurs in the bb_2 of Figure 25 for $l_1 = 1$ since it is not accessed by any predecessor except itself.

Definition 18. *(CONFLICT) This classification occurs when there are multiple reachable paths to a particular basic block and each of these paths has a different effective cache state, which may cause faults or misses depending on the path flow.*

CONFLICT occurs, for example, in Figure 26 in bb_3. If the execution flow is $0 \rightarrow 2 \rightarrow 3$, there is a hit in the cache since $l_1 = 1$ is accessed in bb_2; if the execution flow is $0 \rightarrow 1 \rightarrow 3$, there is a cache miss since there are no references to $l_1 = 1$ in predecessors basic blocks.

After the instruction classifications, we can count the number of faults (A_MISS) that impact directly on the basic block time. The classes F_MISS and CONFLICT are used during the path analysis to determine the program flow that maximizes the execution time.

5.3.2 Pipeline Analysis

The objective of this analysis is to determine the execution time of instructions and basic blocks when executed in the processor pipeline. This analysis does not consider hardware elements like main memory latencies and instruction cache.

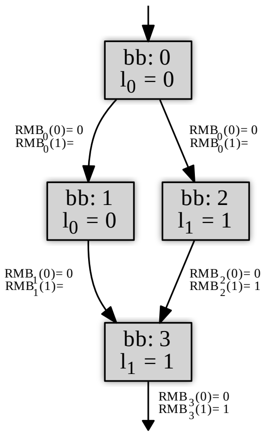


Figure 26 – CONFLICT classification example.

In processors with pipeline, the execution of a single instruction in cycles will be equivalent to the number of pipeline stages. However, this time may be higher if any hazard occurs involving data dependencies of previous instructions. Considering that the used architecture has 5 stages of pipeline it would take 5 cycles for an instruction to be executed. If there were two instructions, the total time will be 6 cycles, and so forth. Algorithm 5 shows our approach to calculate the execution time of a specific basic block.

In this algorithm, represented by the *calculateBasicBlockTime* function, we compose the basic time (number of bundles) with the number of extra cycles generated by certain instructions in these bundles. Table 5 shows instructions that need more than one cycle to execute. For multicycle operations (Line 10), extra cycles are calculated separately for instructions that execute when the predication bit is true or false, because these extra cycles overlap in the bundle execution, in other words, they are not cumulative. For control flow operations (Line 13) the instructions latency is added directly to the basic time, because these instructions are unconditionally executed, independently of the value of the predication bit.

Between bundles may occur data hazards (Line 17) due to data

Algorithm 5 Basic block timing calculation

```
1: function CALCULATEBASICBLOCKTIME($bb$) ▷ Procedure to calculate the execution time of a basic block
2:     $bundles \leftarrow getBundles(bb)$
3:     $basicTime \leftarrow |bundles|$
4:     $extraTime \leftarrow \{0,0\}$
5:     $prevBundle \leftarrow nil$
6:     for all $b \in bundles$ do
7:         $operations \leftarrow getOperations(b)$
8:         for all $op \in operations$ do
9:             $pred \leftarrow getPredicationBit(op)$
10:            if $isMulticycle(op)$ then
11:                $extraTime[pred] \leftarrow getCycles(op)$
12:            end if
13:            if $isControlFlow(op)$ then
14:                $basicTime \leftarrow basicTime + getCycles(op)$
15:            end if
16:            if $prevBundle \neq nil$ then
17:                if $hasHazard(op, prevBundle)$ then
18:                    $extraTime[pred] \leftarrow hazardCycles(op)$
19:                end if
20:            end if
21:        end for
22:        $prevBundle \leftarrow b$
23:    end for
24:    $basicTime \leftarrow basicTime + max(extraTime[0], extraTime[1]) + pipelineLength - 1$
25:    return $basicTime$
26: end function
```

Table 5 – Cycles for different types of instructions.

Operation	isMulticycle	isControFlow	Cycles
Multiplication	x		3
Division	x		19
Memory	x		2
Call		x	4
Goto		x	4
Branch		x	4

dependencies. Table 6 shows when we can have stalls due to hazards for operations in adjacent bundles. For example, if we have a comparison operation followed by a conditional branch that depends on the result of such operation, then we will have a stall. As multicycle operations, the calculation are stored separately for different values of the predication bit.

Table 6 – Hazards between operations.

First op. \ Second op.	Branch	Goto with register
ALU operation	1	0
Memory load	0	1

The final execution time is given by Line 24, which the maximum extra time of the different values of the predication bit. However, to model the execution flow we should apply a correction similar to (ENGBLOM, 2002), as shown in Figure 27. In this figure, we want to get the execution time of the flow between basic blocks 1 and 2 with times 8 and 6 cycles respectively. The sum of the execution times of both blocks is 14 cycles and this does not represent the real execution time of the flow. The correct execution time is 10 cycles as shown in Figure 27 due to "an amendment" in the pipeline between both basic blocks. Thus, during the flow analysis and WCET obtaining, we must subtract $\delta = 4$ cycles for each edge transition between basic blocks. This correction is applied directly in the problem formulation using integer linear programming for obtaining the WCET of the entire program, as we shown in the following subsection.

Although, we can only use $\delta = 4$ when we have two sequential basic blocks without any branch instruction between them. As control flow operations always penalize the execution by 4 cycles, we must use the appropriated delta for each type of successor. Table 7 shows the penalization and δ value for each type of successor. For example, if we have a *Not taken successor*, there is no penalization, then we must use a $\delta = 8$, to compensate the $pipelineLength - 1 + getCycles(Branch) = 8$ portion of the first basic block timing.

5.3.3 Worst-case path search

We previously referenced IPET (LI; MALIK, 1995) (OTTOSSON; SJODIN, 1997) as being an efficient technique to search worst-

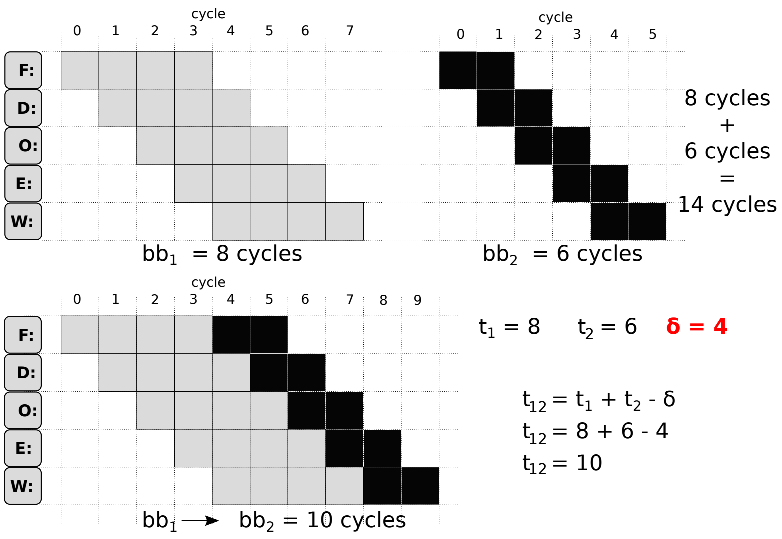


Figure 27 – Example of timing composition of two successive basic blocks

Table 7 – Penalties for different types of flow transfers. In *Direct* flow, does not exists control flow instructions between the considered basic blocks.

Flow type	δ	**Penalty**
Direct	4	0
Not taken successor	8	0
Taken successor	4	4
Not taken successor (predicted)	7	1
Taken successor (predicted)	2	2
Call successor	4	4
Jump successor	4	4
Return successor	4	4

case paths. In this part we present the modeling of linear constraints made in the context of the implemented analyzer. The modeling follows an approach similar to (LI; MALIK, 1995).

We consider that each basic block bb_i is executed x_i times with execution time t_i for modeling the optimization problem. Then the objective function is to maximize equation (Equation 5.2):

$$obj = maximize \sum_{\forall bb_i} x_i \times t_i \tag{5.2}$$

Note that, with the approach used to obtain the basic block times, this estimate becomes pessimistic. Such analysis considers the time from the first instruction of the basic block entering the pipeline until the exit of the last instruction. However, the execution of successive basic blocks is amended within the pipeline, as showed in the previous subsection. This can be expressed in ILP as a discount of δ each time a basic block is executed. Then we have (Equation 5.3):

$$obj = maximize \sum_{\forall bb_i} x_i \times t_i - x_i \times \delta \tag{5.3}$$

We can rewrite the previous equation considering the edges of the CFG, instead of nodes (basic blocks) to facilitate the modeling of execution contexts. Given that the execution time of a basic block x_i can be rewritten using edges as $x_i = \sum_{\forall bb_j \to bb_i} dj\_i$. We get the final objective as described by Equation 5.4.

$$obj = maximize \sum_{\forall bb_i} \left(\sum_{\forall bb_j \to bb_i} dj\_i \times t_i - dj\_i \times \delta \right) \tag{5.4}$$

For the example of Figure 24 we obtain the objective (in GNU MathProg) showed in Code 5.4.

Code 5.4 – IPET problem objective in MathProg for the example of Figure 24

```
maximize wcet: d7_0*14 - d7_0*4 +
d4_1*9 - d4_1*4 +  d5_1*9 - d5_1*4 +
d0_2*9 - d0_2*4 + d1_2*9 - d1_2*4 +
d2_3*9 - d2_3*4 + d3_4*15 - d3_4*4 +
d3_5*15 - d3_5*4 + d2_6*8 - d2_6*4 +
dstart7*7 - dstart7*4 + d6_8*5;
```

5.3.3.1 ILP Constraints

The objective function presented above requires linear constraints to operate, otherwise it cannot converge. The IPET technique consists of a set of constraints which mainly consider flow conservation and loop bounding. According to IPET, the following restrictions shall be applied:

Start and end of execution constraint: all program execution must have a beginning and an end. So the flow must pass exactly once by the CFG entry and exit nodes, which are represented in the CFG as *dstart* and *dend* respectively. The constraint is modeled by Equation 5.5.

$$dstart = 1 \quad \& \quad dend = 1 \tag{5.5}$$

Flow conservation constraint: all flow entering a basic block from a predecessor, should come out from a successor. The flow must be maintained during the execution of the program. This restriction is modeled by Equation 5.6, which must be valid for each basic block bb_i.

$$\sum_{\forall bb_j \rightarrow bb_i} dj\_i - \sum_{\forall bb_i \rightarrow bb_k} di\_k = 0 \tag{5.6}$$

For the example of Figure 24 we extract the control flow conservation constraints presented in Code 5.5.

Code 5.5 – Control flow conservation constraints for the example of Figure 24

```
s.t. xc0: d7_0 - d0_2 = 0;
s.t. xc1: d4_1 + d5_1 - d1_2 = 0;
s.t. xc2: d0_2 + d1_2 - d2_3 - d2_6 = 0;
s.t. xc3: d2_3 - d3_4 - d3_5 = 0;
s.t. xc4: d3_4 - d4_1 = 0;
s.t. xc5: d3_5 - d5_1 = 0;
s.t. xc6: d2_6 - d6_8 = 0;
s.t. xc7: dstart7 - d7_0 = 0;
s.t. xc8: d6_8 - dend8 = 0;
```

Loop bound constraint: basic blocks should execute according to the bounds of the loops to which they belong. A basic block can belong to several loops, provided that they are nested. If a basic block is the header of a loop, it can execute once more at the end to test the exit condition. For limitation of loops, the constraint of Equation 5.7 must be valid for each basic block bb_i with loop bound lb_i.

$$\sum_{\forall bb_j \to bb_i} dj\_i <= lb_i \tag{5.7}$$

For the example of Figure 24 we derived loop bound constraints presented in Code 5.6.

Code 5.6 – Loop bound constraints for the example of Figure 24

```
s.t. x0: d7_0 <= 1;
s.t. x1: d4_1 + d5_1 <= 5;
s.t. x2: d0_2 + d1_2 <= 6;
s.t. x3: d2_3 <= 5;
s.t. x4: d3_4 <= 5;
s.t. x5: d3_5 <= 5;
s.t. x6: d2_6 <= 1;
s.t. x7: dstart7 = 1;
s.t. x8: d6_8 <= 1;
```

Loop execution constraint: a loop is an auto conservative or strongly connected component, it only shall be considered part of the WCET if it is effectively executed. A loop is executed when the flow enters its header. Equation 5.8 ensures this constraint.

$$\sum_{\substack{\forall bb_j \to bb_i \\ \wedge bb_j \notin loop(bb_i)}} dj\_i \times ilb_i - \sum_{\substack{\forall bb_i \to bb_k \\ \wedge bb_k \in loop(bb_i)}} di\_j = 0 \tag{5.8}$$

The previous equation must be valid for every basic block bb_i which is loop header. ilb_i represents the bound of the loop of which this basic block is header, and $loop(bb_i)$ represents the respective loop. In this constraint, the edges that come from the header to the inner loop blocks can be traversed by the flow if this flow comes from an outer edge of the loop. According to the definition of (LI; MALIK, 1995), these restrictions may also be classified as *Program Execution Constraints*.

Considering the example of Figure 24 we can obtain loop the loop execution constraint presented in Code 5.7.

Code 5.7 – Loop execution constraint for the example of Figure 24

```
s.t. xa12: d0_2*5 - d2_3 = 0;
```

With the above constraints, we can obtain the solution to the problem of the example. Figure 28 graphically shows the result. In this figure, the black edges belong to WCEP (*worst-case execution path*) and are computed on WCET. Moreover, the gray edges are not part of WCEP. This means that in the worst case, the flow will not pass through the basic block 5.

In the example of Figure 28, the cache memory was not yet considered. In order to estimate its influence, new ILP constraint must be developed as described further.

Instruction cache constraints: the WCET of a program must take into account that the flow through certain basic blocks can be impacted by different cache behaviors. Another factor is that now the execution of basic blocks is context dependent: a basic block C may have distinct execution times when reached from the predecessors A or B. To represent this effect we associate weights of edges with the execution times of basic blocks when reached by these edges.

Our cache modeling follows, in general, the technique proposed by (LI et al., 1996), except that the modeling presented here takes into account only CFG edges as variable/problem entities. As the IPET modeling considers only edges, we simply associate the weights of these with the execution times of the basic blocks when reached by such edges.

As an example, consider the CFG of Figure 29 with the instruction cache. Basic block 1 can have one execution time when reached by the basic block 4 and another time when reached by the basic block 5, due to distinct cache states.

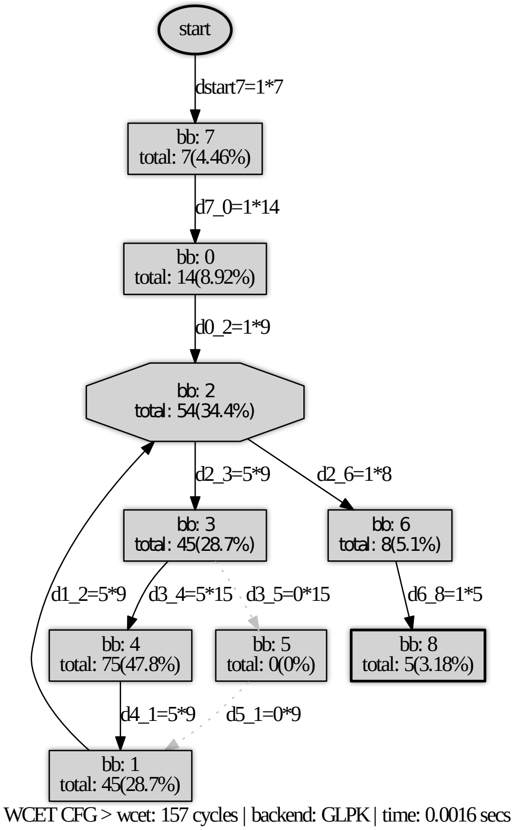


Figure 28 – IPET result

From the three considered cache states, only one of them needs special attention in the IPET model, which is the *first miss*. First miss occurs only once for each basic block that has it, each time the loop which contains this basic block is executed. So, all first misses in a loop occur in the first iteration of each complete execution of this loop. To address this, the CFG is expanded into a *multigraph*, and the *first miss* modeled as a separate edge. With a dedicated edge, it is possible to model constraints that control the occurrence of this event. The formal definition of the Control

Flow Multigraph is given as follows:

Definition 19. *(Control Flow Multigraph) A Control Flow Multigraph is a directed graph $G = (V, E, i)$, where the nodes (V) represent basic blocks and edges are defined by $d_{tipo} \in E \subseteq V \times V$, as the previous definition of control flow graph. If there is a first miss between pairs of basic blocks, then a pair of edges between the two blocks exists, one representing the first miss d_{fm} (type $= fm$) and another representing flows of subsequent iterations (hits) d_h (type $= h$). The third edge type represents all other situations (always hit, always miss and conflict). The weights of edges $W(di\_j)$ represent the cost to execute the basic block j, when preceded by the basic block i, considering cache effects.*

In Figure 29, one can see that will occur a first miss in the basic block 4 represented by the thicker edge.

The "fm" edges means *first miss* and "h" represents the further flow transfers, where occur *cache hits*. We do not need a special treatment for *always-hit*, *always-miss* and *conflict*, since the resulting time for the reached basic blocks is always fixed when it is succeeded by a given predecessor.

For modeling first misses, we must follow different approaches for basic blocks that are loop headers and blocks that are not:

- In loop header, a first miss (if any) will occur only by the outer edge of the loop. The constraint is modeled by Equation 5.9:

$$\sum_{\forall bb_j \to bb_i \wedge bb_j \notin loop(bb_i)} dj\_i \leq elb_i \tag{5.9}$$

  elb_i is the bound of the outer loop bb_i.

- For normal basic blocks the constraint is modeled by Equation 5.10, for each bb_i:

$$\sum_{\forall bb_j \to bb_i} dj\_i \leq elb_i \tag{5.10}$$

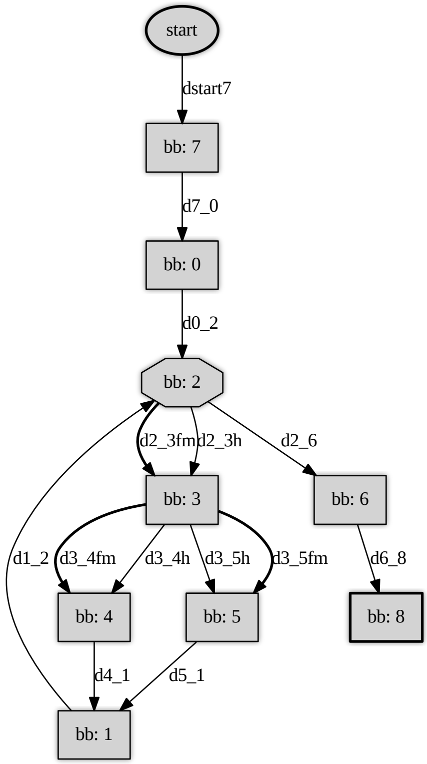


Figure 29 – Multigraph of the example

For the used example, we can extract the cache constrains presented in Code 5.8.

Code 5.8 – Cache constraints for the example of Figure 29

```
s.t. xcache3: d2_3fm <= 1;
s.t. xcache4: d3_4fm <= 1;
s.t. xcache5: d3_5fm <= 1;
```

By joining all the previous constrains, we can calculate the WCET for the previous example. The result is shown in Figure 30. One can see that the worst-case execution time for the analyzed program is 421 processor cycles, and the result was obtained in 0.00176 seconds. The solver used by our tool for solving the problem is GLKP (MAKHORIN, 2008).

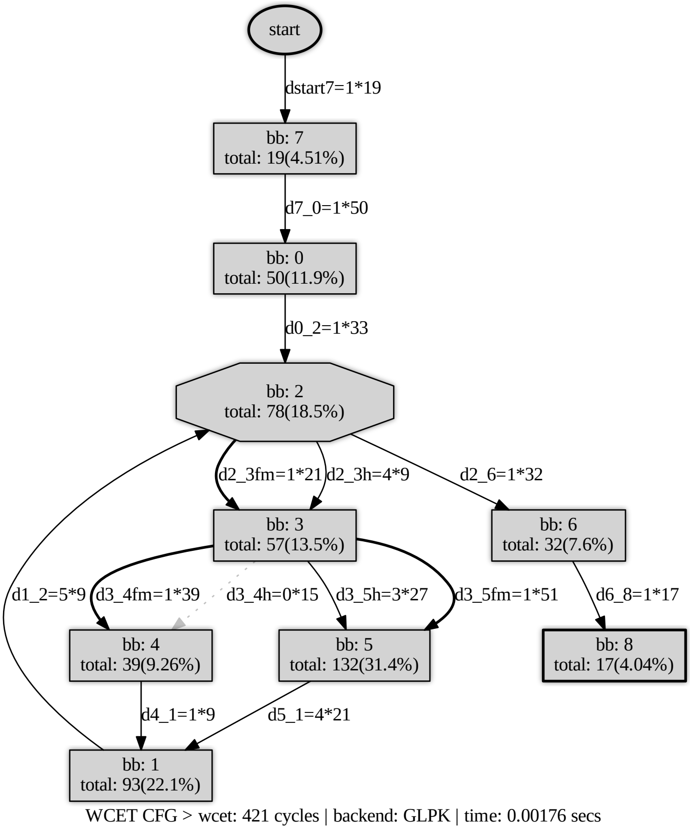


Figure 30 – IPET result considering cache

5.4 ENABLING WCET REDUCTION SCHEMES

It is difficult to perform WCET-oriented optimization using LLVM due to its highly optimized passmanager that isolates the treatment of each function of a compilation unit. Due to this fact, we cannot optimize the program as a whole aiming at WCET reduction using the standard LLVM pass manager because the generated code is only fully materialized at the end of the complete process. Moreover, the pass manager deallocates any machine related code representation structure of a function after writing its generated object code to file at the end of the pass manager execution. So, when we can finally calculate the WCET of a program, we cannot use this data to change the code (optimization application), because the needed intermediate structures no longer exist.

To apply WCET reduction techniques, or any technique that relies on this type of information, there must be an integration of the compiler with some WCET analyzer for identification of potential optimization points. The way the LLVM was built complicates some aspects of the application of WCET reduction techniques, as follows:

- Impossibility to undo changes in an automated manner. When applying an optimization involving WCET reduction, we may test if this application is effective in reducing WCET, and if so, the optimization is maintained, otherwise it will be undone. In LLVM there is no simple way to reverse code changes. There is no way to copy the data structures and objects to save an specific state of a program being compiled.

- Independence in the treatment of functions. When the LLVM pass manager is processing a function, specifically applying passes for the code generation, there is no information available on the generated code of other functions. This happens in two scenarios:

  - A function of interest (other than the current, in processing) has been processed and the code generated for this is already stored at the final object code, with all the intermediate structures which have been used deallocated by the manager. This occurs because the LLVM adopts a mechanism of passes that automatically releases all data relative to a given function, after the execution of all passes over this.

- A function of interest has not yet been touched by the pass manager. The data has not been generated to be deallocated.

The problem of those characteristics to reducing WCET is that we can only see one function at a time (in machine code). The application of optimizations for WCET requires a view of the whole program in terms of generated code. It is not possible to calculate the WCET of an isolated function because it depends on the context in which this can be called and times of functions called within the function. For example, a function *A* can contain a *if-then-else* sentence, where the branch *then* calls a function *B* and the branch *else* another function *C*. In this case, it is impossible to know whether the WCEP of function *A* follows the branches *then* or *else* if we do not know the WCET of the called functions.

Due to this fact, strategies like that proposed by (FALK et al., 2006), where the analyzer is invoked directly by the compiler to take optimization decisions cannot be used. Another approach is presented by (PUSCHNER et al., 2013). In this approach, the compilation is coordinated with the WCET analysis by a higher level planning tool. In the next subsections, we describe how we worked around those issues.

5.4.1 Approach 1: Back-end adaptation To Use WCET Information

To conduct WCET-oriented optimization, the compiler back-end and the WCET analyzer where integrated as proposed by (FALK et al., 2006). LLVM has been modified to not destroy the objects and data structures for the machine code generated at the end of the pass manager execution. This information is retained until the moment of destruction and deallocation of intermediate representation as a whole, which is the module and its functions. This allows us to make any changes to any function and regenerate the object code file, at the end of all LLVM code generation passes. With this LLVM adaptation, any change at the machine code level can be done in any function, followed by an entire object code generation, many times as needed. Using the previously explained LLVM modification with a WCET tool integration, we can perform any WCET-oriented optimization in an iterative way.

5.4.2 Approach 2: Code Optimizations Guided by an External Planning Tool

The second approach is similar to the (PUSCHNER et al., 2013) approach. In this approach, a tool in a higher or planning level is responsible to select the parts of the program that must be optimized, using WCET information as guidance. This tool shares a database with the compiler that is used as communication channel. This database stores facts about the structure of the program and values that specify if such structure must be touched by a specific optimization. The tool invokes the compiler to generate the object code and data used as input for the WCET analyzer. After that, WCET information is obtained through the WCET analyzer. Using this information, the planning tool updates the database using optimization heuristics. This task repeats until WCET stabilization or when the entire code is already analyzed by the planning tool.

Using this strategy, we can perform any WCET-oriented optimization in an iterative way. Optimizations must keep consistency between the transformed code and the annotations provided in the source. Figure 31 shows a simplified diagram of our tools and their connection. As we can see, both LLVM infrastructure and planning tool share a data-base containing information about the program structure, which is empty at the first program compilation. From the first compilation, the planning tool can invoke the WCET analyzer tool and execute strategies to guide the optimization process.

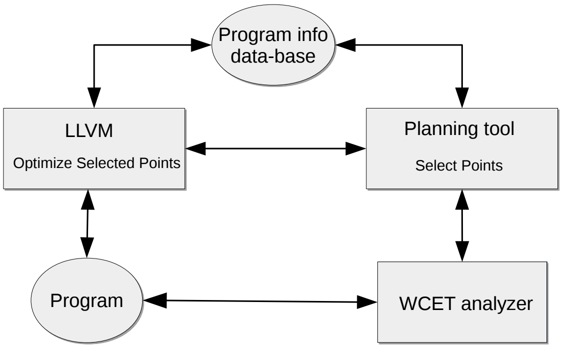


Figure 31 – Diagram representing the tools that compose the infrastructure used

5.5 CHAPTER SUMMARY

This chapter presented the tools developed to compose the experimental infrastructure used in this work. This chapter also briefly described the target architecture used in our work. The first tool that we presented was the compiler, with its packetizing and alignment passes. We also incorporated into the compiler a pass for extracting the control flow graph used in the second tool, which is WCET analyzer. The backend has approximately 25k lines of code.

The second tool developed is responsible for calculating the worst-case execution time of programs. This tool has input, the control flow graph and annotations representing worst-case iteration counts of loops. The development of this tool has required the study and implementation of various techniques related to obtaining WCET such as those presented in (LI; MALIK, 1995), (ENGBLOM; ERMEDAHL, 2000). This tool has approximately 10 thousand lines of C++ code. The tool has been implemented completely from scratch, and the only library used is the integer linear programming solver GLPK.

The last part of this chapter presented how we connected the compiler and the WCET analyzer in order to enable WCET-aware optimizations.

6 CONTRIBUTION 1: COMBINING LOOP UNROLLING STRATEGIES AND CODE PREDICATION

The goal of this chapter is to propose a different way to perform loop unrolling on data-dependent loops using code predication targeting WCET reduction, because existing techniques only consider loops with fixed execution counts. We also combine our technique with existing unrolling approaches. Results showed that this combination can produce aggressive WCET reductions when compared with the original code. This contribution was firstly published in (CARMINATI et al., 2017).

6.1 INTRODUCTION

Loops are frequently good target candidates for compiler optimizations to extract performance of modern processor architectures. Loop unrolling is a well-known technique used to improve average-case performance of programs. This technique consists in replicating the loop body for a certain number of times to avoid branch and jump overhead and to reduce the number of increment/decrement operations, inserting extra code to verify exiting corner cases, if necessary. The number of body replications is often called *unrolling factor* and the original loop is called *rolled loop*.

Loop unrolling can contribute to improve the instruction level parallelism (ILP) and execution performance of programs, by enabling more optimization that are affected by code expansion. Although, this code expansion can lead to instruction-cache performance degradation, if not carefully applied. If loop unrolling is applied before the register allocation phase, register pressure can be increased, leading to the insertion of more spill and reload operations in the code. However, a standard compiler cannot use loop unrolling directly if worst-case execution time (WCET) reduction is desirable, due to the instability of the execution path that generates the worst possible execution time and negative cache effects. Some techniques were proposed in the literature to achieve WCET reduction using loop unrolling, as in (ZHAO et al., 2006) and (LOKUCIEJEWSKI; MARWEDEL, 2010). In these works, loops are carefully unrolled to promote WCET reduction and limit code increase. But, only loops with fixed execution counts are considered.

The contribution of this chapter is twofold. Firstly, we propose an alternative way to perform loop unrolling on loops with arbitrary (or variable) execution counts. Traditionally, loops with unknown execution counts are unrolled with fixed unrolling factors, with the corner conditions (i.e, the unrolling factor is not a multiple of the number of iterations) treated with branch instructions. The approach adopted in this work is to treat the same corner conditions using code predication instead of instructions that perform control flow changes. Code predication is already used in software pipelining of loops, but its application directly with loop unrolling was not reported in the literature. Code predication also can be explored using a transformation called *If-Conversion*, which is a standard compiler optimization that converts control dependencies into data dependencies, removing branches.

The second contribution of this chapter is the combination of our technique with other standard unrolling approaches for data dependent loops and loops with fixed execution counts. In this way, we can decide on a per loop level which of the approaches should be used for loop unrolling. This combination of techniques is important because not every loop can be unrolled in the same way. For example, loops with variable number of iterations must include compare and branch instructions to treat different exit conditions, but loops with static execution counts can be unrolled without these instructions. The necessity of compare and branch instructions is not the only difference when unrolling these two types of loops, but the selection of a valid unrolling factor is also different. In loops with a static number of iterations, we can only consider unrolling factors that perfectly divide such number of iterations. Until the present moment, no work addressing the combination of different unrolling techniques was identified in the literature.

The remainder of this chapter is organized as follows: Section 6.2 outlines the related work on loop unrolling directed to WCET reduction. Section 6.3 shows the motivations of this chapter. Section 6.4 explains the proposed approach to perform loop unrolling targeting real-time applications. In Section 6.5 we describe briefly our testbed. Section 6.6 presents the obtained results using a benchmark suite. Section 6.7 presents our conclusions and final remarks.

6.2 SUMMARY OF RELATED WORK

The first work that concerns WCET reduction using loop unrolling, consists in applying this optimization directly at assembly level (ZHAO et al., 2006). In this work, only innermost loops with fixed number of iterations are unrolled and the unrolling factor used for all loops is 2. Although, not all candidate loops are unrolled, but only those that are present in the worst-case execution path (WCEP), and they are kept unrolled only if WCET reduction is achieved. At every optimization application, the WCET information must be re-calculated to update the worst-case path information that drives the algorithm. This recalculation is necessary because any code change that affects the WCET may result in a WCEP change. These WCET recalculations are a common strategy employed by compilers focused in worst-case execution time reduction. This technique is explained in Section 4.3.2 of Chapter 4.

Another approach to perform loop unrolling aiming at WCET reduction was proposed in (LOKUCIEJEWSKI; MARWEDEL, 2010). Here, the optimization is applied at the source code level and uses a processor with instruction cache and scratchpad memory. As the optimization is applied at the source code level, the success of next optimizations performed by the compiler is enhanced, specially for those that benefit from code expansion. The key aspects of the technique are: (1) choose the most profitable loops concerning WCET reduction and (2) calculate an unrolling factor considering memory constraints. Consequently, the algorithm balances memory utilization and WCET reduction. This technique is explained in Section 4.3.1 of Chapter 4.

Both the previously presented approaches consider only loops with fixed number of iterations. In fact, both techniques can be used to unroll loops with arbitrary counts or data-dependent loops, providing necessary code to exit the loop when the termination condition is reached. This code is commonly generated as branch instructions.

If-Conversion (ALLEN et al., 1983) is a technique used to convert control dependencies into data dependencies. The basic principle consists in eliminating gotos and branches and inserting logical variables to control the execution of instructions in the program. *If-Conversion* can be performed at IR-level or machine-level as stated by (JORDAN et al., 2013) and is related to region enlargement techniques used to expand the instruction scheduling scope beyond a single basic block, which is specially beneficial for *very long instruction word machines* (VLIW).

The application of *If-Conversion* techniques in loops is not a novel idea. Software pipeline (CHARLESWORTH, 1981) can benefit from *If-Conversion* and code predication to control the execution of prologue and epilogue of pipelined loops (DEHNERT et al., 1989). Another technique that can benefit from *If-Conversion* is loop flattening (HANXLEDEN; KENNEDY, 1992). Loop flattening is a form of software pipelining that merges nested loops into a single loop body, providing necessary code to control the execution and the flow of data between blocks. In (POP et al., 2010) *If-Conversion* is used to eliminate back-edges of flattened loops. The next section outlines the motivation and the key ideas behind the proposed unrolling technique.

6.3 MOTIVATION

We can consider the loop of Code 6.1 as a motivational example. This code shows a loop with the number of iterations dependent on the value of a variable (called data-dependent loop). For this loop, a compiler commonly generates a control flow structure that is shown in Figure 32. In this structure, a simple *for* loop has two basic blocks called *header* and *body* which are surrounded by an *entry* and an *exit* basic blocks.

There are some approaches to perform the loop unrolling opti-

Code 6.1 – Simple data-dependent loop.

```
void loop(int a){
    int i, j = 0, k = 0;

    for(i = 0; i < a ; i++){
        j++;
        k++;
    }
}

int main(int a){
        loop(90);
}
```

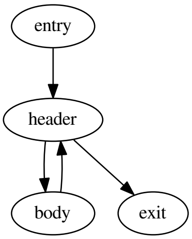


Figure 32 – Control flow graph of Code 6.1.

mization considering this loop. The simpler strategy consists in optimizing only loops with fixed counts. In this case, the compiler chooses an unrolling factor that exactly divides the number of iterations of the loop. If a compiler is able to optimize data-dependent loops with unknown number of iterations, it must take care of left-over iterations. Another problem with data-dependent loops is the difficulty to choose an effective unrolling factor.

Code 6.2 shows the application of loop unrolling on the data-dependent loop of Code 6.1 (in C code for simplicity). Is this case, if the compiler is not able to calculate the number of iterations for

Code 6.2 – Unrolled loop.

```c
void loop(int a){
    int i,j = 0,k = 0,L = 0;

    for(i = 0; i < a; i+=L){
        L = 1;
        j++;
        k++;
        if(i+L >= a) break;
        j++;
        k++;
        L++;
        if(i+L >= a) break;
        j++;
        k++;
        L++;
    }
}
```

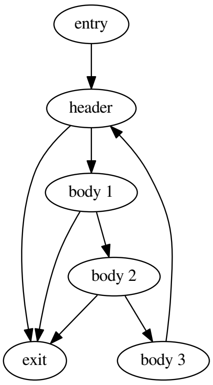


Figure 33 – Control flow graph of Code 6.2.

the loop, or determine whether this number is odd or even, it must check the exit condition on every body replication, as done by the *if* statements. This condition checking leads to a control flow graph that is shown in Figure 33.

Note that with this approach, the number of branching instructions is augmented, also increasing the number of basic blocks. From the WCET perspective, by increasing the number of basic blocks, we increase the search space that contains the worst-case execution path that produces the WCET. From the code generation point of view, a branch may need up to 3 intermediate operations that are not necessarily in this order: (1) condition calculation, (2) target address calculation and (3) branch execution. Depending on the target architecture, all previous operations are executed by one instruction or are segmented in sequences of 2 or 3 instructions.

Considering the use of a target architecture with instruction predication support, it is possible to remove branch operations (if any) from loops that are unrolled. For this purpose, we can consider the loop of Code 6.3 and its respective CFG shown in Figure 34. This loop is semantically equivalent to the loop of Code 6.2. If we can rewrite an unrolled loop in terms of conditional expressions, as done to Code 6.2 to obtain Code 6.3, it is possible to apply *If-Conversions* to the code. Until the writing of this work, no technique exist in the literature to perform these transformations to unrolled data-dependent loops. As we stated before, *If-Conversion* is an optimization technique that converts control dependence into data dependence through the definition of guards to control the execution of instructions. If the target architecture supports instruction predication, *If-Conversion* can result in branchless code, reducing code size and number of basic blocks. Generically, an application of *If-Conversion* to the code of Code 6.3 would produce the control flow graph of Figure 35. The prefix *(p)* means that the execution of the basic blocks *body 2* and *body 3* are conditioned to some predication guard *p*.

From the WCET perspective, the common behavior of analyzers is to consider the complete execution of the loop iterations. In this way, the last iteration will be considered fully executed, even with the possibility of an early loop exit if the condition is reached. If a loop is always fully executed in the worst case, it is beneficial to reduce the number of instructions of the unrolled loop, and if premature exits

Code 6.3 – Unrolled loop rewritten with conditional expressions.

```c
void loop(int a){
    int i, j = 0, k = 0;

    for(i = 0; i < a;){
        j++;
        k++;
        i++;

        if(i < a){
            j++;
            k++;
            i++;
        }
        if(i < a){
            j++;
            k++;
            i++;
        }
    }
}
```

will never be taken (branch instructions), we can eliminate them from the code using predication. Using code predication, we decrease the number of instructions while preserving the semantics of the code.

In the next section, we present our approach to perform loop unrolling which applies simultaneously code predication directly in machine code. The technique starts from a simple data-dependent loop and directly generates an unrolled and predicated version, as done step-by-step in this section. The main improvement of our approach is that it avoids the use of branch instructions, differently from what is usually done by traditional techniques.

6.4 OUR LOOP UNROLLING APPROACH

Our loop unrolling algorithm performs code predication in conjunction with the unrolling steps. In this way, sophisticated *If-Conversion* strategies can be avoided. The algorithm must be used directly in as-

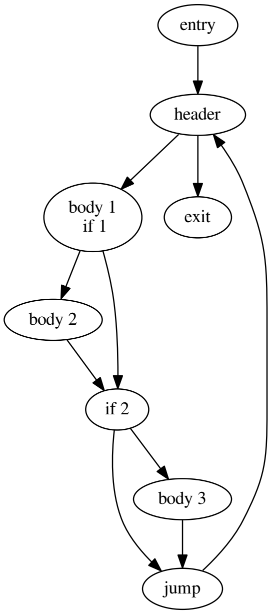


Figure 34 – Control flow graph of Code 6.3.

sembly representation. The architectural requirement of the technique is the existence of full-predication mechanisms to control the execution of instructions. As example of such mechanisms, we can cite IA-64 (GEVA; MORRIS, 1999) and ARM (FURBER, 1996) (except for Thumb instructions). The technique also benefits from branches that are segmented in sequences of more than one operation.

The steps to unroll a loop are shown by Algorithm 6. The algorithm assumes that every loop that will be unrolled is composed by a header and a body. This constraint must be ensured by the caller of the algorithm procedure. Another requirement is the implementation of loop headers with compare instruction followed by branch instructions to control the loop exit.

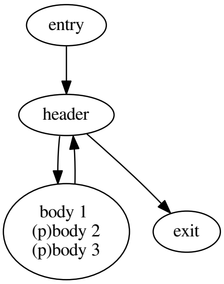


Figure 35 – Control flow graph representing an *If-Conversion* of the code from Code 6.3.

Algorithm 6 Predicated Loop Unrolling algorithm.

```
 1: procedure PREDICATEDLOOPUNROLLING($Loop$, $U$, $P$)▷ Unroll loop u times
 2:     $Header \leftarrow loopHeader(Loop)$
 3:     $Body \leftarrow loopBody(Loop)$
 4:     $removeUncondBranch(Body)$
 5:     $BodyCopy \leftarrow createCopy(Body)$
 6:     for $i \leftarrow 1$ to $U - 1$ do
 7:         $NewHeader \leftarrow createCopy(Header)$
 8:         $removeConditionalBranch(NewHeader)$
 9:         $changeCompareOutput(NewHeader, P)$
10:         $Body \leftarrow unify(Body, NewHeader)$
11:         $NewBody \leftarrow createPredCopy(BodyCopy, P)$
12:         $Body \leftarrow unify(Body, NewBody)$
13:     end for
14:     $insertUncondBranch(Body, Header)$
15: end procedure
```

The algorithm works as follows: First, header and body are identified, which is done by Lines 2 and 3. The second step removes the unconditional branch from the loop body to the header. This branch instruction will be re-inserted at the end of the algorithm, as a last instruction (Line 14). The next step is to unroll the loop using the provided unrolling factor, using the original loop body as first copy.

For each unroll step, we create a copy of the header, converting control flow instructions into instructions that control the predication of subsequent copies of the loop body (Lines 7, 8, 9 and 10). Then, Code 6.4 – Example of loop in assembly code.

```
 1     add $r8 = $zero , 0
 2     add $r9 = $zero , 0
 3     add $r10 = $zero , 0
 4  HEADER:
 5     cmplt $br0 , $r9 , $r16
 6     brf $br0 , $EXIT
 7  BODY:
 8     add $r10 = $r10 , 1
 9     add $r8 = $r8 , 1
10     add $r9 = $r9 , 1
11     goto  $HEADER
12  EXIT:
13
14
15
16
17
18
```

Code 6.5 – Unrolling using code predication.

```
 1     add $r8 = $zero , 0
 2     add $r9 = $zero , 0
 3     add $r10 = $zero , 0
 4  HEADER:
 5     cmplt $br0 , $r9 , $r16
 6     brf $br0 , $EXIT
 7  BODY:
 8     add $r10 = $r10 , 1
 9     add $r8 = $r8 , 1
10     add $r9 = $r9 , 1
11     cmplt $p , $r9 , $r16
12     (p)add  $r10 = $r10 , 1
13     (p)add  $r8 = $r8 , 1
14     (p)add  $r9 = $r9 , 1
15     goto  $HEADER
16  EXIT:
17
18
```

Code 6.6 – Unrolling using the standard approach.

```
 1     add $r8 = $zero , 0
 2     add $r9 = $zero , 0
 3     add $r10 = $zero , 0
 4  HEADER:
 5     cmplt $br0 , $r9 , $r16
 6     brf $br0 , $EXIT
 7  BODY0:
 8     add $r10 = $r10 , 1
 9     add $r8 = $r8 , 1
10     add $r9 = $r9 , 1
11     cmplt $br0 , $r9 , $r16
12     brf $br0 , $EXIT
13  BODY1:
14     add $r10 = $r10 , 1
15     add $r8 = $r8 , 1
16     add $r9 = $r9 , 1
17     goto  $HEADER
18  EXIT:
```

we make a predicated copy of the body that is amended at the end of the original body (Lines 11 and 12). The algorithm basically removes forward branches used to exit the loop and inserts boolean guards to control the execution of the remaining part of the loop. These guards are stored in the P variable.

It is relevant to notice that the first copy of the loop body does not need to be predicated, because the header condition verification ensures that at least one iteration (in relation to the rolled loop) must be executed, otherwise the loop must be already terminated. In this way, the first copy of the body represents exactly the original basic block of the loop.

6.4.1 Example

As an example, the algorithm is applied the Code 6.1. The assembly code dialect used is referent to the ST231 ISA, which is also used in our testbed. We omit bundles delimitation in code listings for simplicity. Before the unroll, the sequence of instructions generated is shown in Code 6.4.

After the application of the algorithm and using an unrolling factor of 2, we obtain the code as shown in Code 6.5. The sequence *(p)* means that the operation execution is conditioned to the content of the flag register p, which is a common notation of predicated code. For comparison purposes, the same code is unrolled in the standard way as shown in code listing of Code 6.6. Comparing the two approaches, we can see that the predicated version presented fewer instructions than the standard counterpart (with branches).

6.4.2 Combining Loop Unrolling techniques

As each loop unrolling technique can be applied to a set of loops that share a certain characteristic, it makes more sense to combine the techniques to get a more aggressive WCET reduction, instead of comparing them. In this way, we decide in a per loop level which approach should be applied.

Depending on loop attributes, we consider three unrolling alternatives:

Standard without branches For loops that are not data-dependent (fixed execution counts), we can use the simplest loop unrolling approach. This approach replicates the loop body using an unrolling factor that divides the execution count of the loop. As this approach is a common strategy considering compiler optimization, we will omit its representation in pseudo-code. We will refer to this approach as $simpleLoopUnrolling\ (loop, unrollingFac)$ as the algorithm representation and its parameters.

Standard with branches For data-dependent loops with some kind of control flow change inside of the loop body, we can use loop unrolling with compare and branch instructions to exit the loop when the condition is reached. For simplicity, we apply this unrolling alternative to loops with call instructions in the body. This approach is also a common strategy considering compiler optimization, and we will omit its representation in pseudo-code. We will refer to $branchedLoopUnrolling\ (loop, unrollingFac)$ as the algorithm representation and its parameters if we must use this approach. This approach cannot be used with function inlining, although this is not a problem because we do not use this type of optimization for two reasons: (1) we do not apply any optimization when we cannot quantify WCET effects. (2) with inlining, we lose the one-to-one mapping between the object code and source code, which is necessary to perform WCET calculation.

Predicated For data-dependent loops with simple loop bodies, we can use the predicated version. We cannot use this type of unrolling in loops with call instructions because condition or flag registers are not commonly exposed to the calling conventions used in processors. If we had to save the flag registers, it would be better to use the previous approach. We will call this approach as *predicatedLoopUnrolling*, as presented by Algorithm 6.

Algorithm 7 chooses the adequate unrolling technique through inspection of the loop characteristics. The field $loop.uf$ represents the unrolling factor that must be used for a specified loop. We cannot choose unrolling factors arbitrarily if our objective is WCET reduction. In the next section, we will show how to use WCET information to choose adequate unrolling factors.

Algorithm 7 Optimization algorithm that is executed by the compiler

```
procedure OPTIMIZELOOPS(Program)▷
    LoopList ← getLoops(Program)
    for each loop ∈ LoopList do
        if not loop.isDataDep then
            simpleLoopUnrolling(loop, loop.uf)
        else if loop.hasCall then
            branchedLoopUnrolling(loop, loop.uf)
        else
            predicatedLoopUnrolling(loop, loop.uf, P)
        end if
    end foreach
end procedure
```

In relation to the predication flag that must be given as parameter of Algorithm 6 in Line 9 (of Algorithm 7), the same register can be used to hold all conditions for all loops, because each copy of the loop body must be guarded by only one condition, i.e., that related to the exit condition of the loop, which is updated before the execution of this body copy. In this way, we can pass any register or flag that can be used to predicate instructions. In this algorithm, we consider candidates for loop unrolling: (1) innermost loops and (2) loops composed by two basic blocks header and body, as the example of Figure 32. We use these restrictions to process only small loops, where we can easily achieve gains using loop unrolling.

6.4.3 Ensuring WCET reduction by unrolling factor selection

The previous algorithm is responsible for unrolling the loops of a program using a set of unrolling factors. It is also necessary to choose a unrolling factor for each loop that minimizes the WCET. As we are interested only in verifying the effectiveness of our technique, we are not concerned in choosing an optimal unrolling factor considering code increase and WCET reduction.

We adopted a scheme that tries to iteratively choose an unrolling factor for each loop in the program. If the loop has no impact on the worst-case execution time, i.e. resides outside the WCEP (*worst-case execution path*), it will be kept rolled, otherwise it will be unrolled. The set of unrolling factors will vary according to characteristics of the loop, such as data dependency and parity of execution counts.

If the unrolled loop increases the WCET, then it will be also kept rolled. Otherwise it will be maintained unrolled using the factor that best minimizes the WCET considering the previously considered ones from the set. Each loop is processed exactly once, and after each loop handling the WCET (and WCEP) information must be updated to guide the treatment of the next loops. To verify if a WCET increase occurs, it is necessary to perform a program recompilation and an invocation to the WCET analyzer. We do not reconsider loops in case of path changes, since typically all loops in a program are on the

WCEP, as stated by (LOKUCIEJEWSKI; MARWEDEL, 2010). We only check if the current loop is on the WCEP.

Algorithm 8 presents our approach for selection of unrolling factors. This algorithm is designed to be executed as a complementary part of the compilation process, and can be implemented as a separated tool. Regarding the flow of information point of view, it is necessary the following interactions between the compiler and the algorithm:

- Compiler $\rightarrow$ Algorithm: the compiler must export all information related to all loops that can be unrolled. The information must allow the correlation between the loops and the worst-case execution time related data. Execution counts must be exported as well. In case of data-dependent loops, execution counts can be provided as annotations in the source code, for example. These execution counts are also necessary for the calculation of the worst-case execution time.

- Algorithm $\rightarrow$ Compiler: The algorithm can provide unrolling factors for all loops that were exported for a determined program. If such unrolling factors are not provided, the compiler keeps the loops rolled. To decide which unrolling factor to use, the algorithm uses WCET analysis and loop information.

As we can see, the previous relation between compiler and algorithm forms a cyclic and incremental approach to optimize loops. The parameter of Algorithm 8 is the representation of a compiled program. The first step of the algorithm is to retrieve a list of (exported) loops of the program representation (Line 2) followed by a WCET analysis (Line 3). The main loop of the algorithm iterates over the loop list (Line 4), considering only loops that are in the WCEP (Line 5). Then, we assume that it will be kept rolled (Line 6) if its is not possible to choose an unrolling factor. The next step consists of testing different unrolling factors in the interval $[2, 17]$, which was obtained experimentally. If a loop can be unrolled, we have basically two alternatives to consider an unrolling factor as valid:

Data independent loops for data-independent loops (Line 8), the unrolling factor must exactly divide the execution count of the loop (Line 9), because we do not want to generate instructions to control the reaching of the exit condition inside the replicated copies of the body.

Data dependent loops for data-dependent loops (sentence of Line 8 is evaluated to false) we do not test if the unrolling factor divides the execution count, because leftover iterations are treated explicitly by compare and branch or compare and predicate instruction, depending on the approach. However, we use a heuristic approach that considers factors whose parity is equal to the the loop bounds' parity (Line 13).

Note that a loop can have more than one possible unrolling factor from the interval $[2, 17]$. In this case, we will use the one that reduces the WCET most. Experience says that even small unrolling factors can produce instruction cache degradation, so, the interval $[2, 17]$ is able to cover a useful range of unrolling factors for real programs. Although, depending on the compiler used and processor characteristics, these values can be tuned experimentally. If such unrolling factor exists, we recompile the program and test for WCET changes. In case of WCET increase (Line 21), we use the last chosen unrolling factor (Line 22) and skip to the next loop. Otherwise we use the actual factor updating the WCET (Line 24 and 25).

The algorithm only cares about data dependency and parity of execution counts to choose unrolling factors. The final decision about which unrolling approach must be applied to data-dependent loops is left to the compiler that implements Algorithm 7.

The time complexity of Algorithm 8 is, at least, $O(n^2)$, where n is the number of loops. We consider at least because we do not know the exactly complexity of the compiler's internal algorithms. In fact, there will be, for any program, one initial invocation to the analyzer to estimate the WCET and other invocations for each loop to test the considered unrolling factors, giving a total of $1 + 16 \times n$ invocations to

Algorithm 8 Algorithm that defines unrolling factors for all optimizable loops in *Program*.

```
 1: procedure CALCULATEUNROLLINGFACTORS(Program) ▷ Algorithm executed by the optimization planning tool
 2:     $LoopList \leftarrow getLoops(Program)$
 3:     $wcetData \leftarrow calculateWCET(Program)$
 4:     for each $loop \in LoopList$ do
 5:         if $isInWCEP(loop, wcetData)$ then
 6:             $lastUF \leftarrow 0$
 7:             for $i \leftarrow 2$ to 17 do
 8:                 if not $loop.isDataDep$ then
 9:                     if not $divides(loop.bound, i)$ then
10:                         $continue$
11:                     end if
12:                 end if
13:                 if $parity(loop.bound) = parity(i)$ then
14:                     $loop.uf \leftarrow i$
15:                     $recompile(Program)$
16:                     $newWcet \leftarrow calculateWCET(Program)$
17:                     if $newWcetData.value >= wcetData.value$ then
18:                         $loop.uf \leftarrow lastUF$
19:                     else
20:                         $wcetData \leftarrow newWcetData$
21:                         $lastUF \leftarrow i$
22:                     end if
23:                 end if
24:             end for
25:         end if
26:     end foreach
27: end procedure
```

the analyzer. The worst case occurs when all loops are data independent and can be divided by factors in the interval $[2, 17]$. In practice, this situation only occurs when the loop count represents a common multiple of all considered unrolling factors. For each loop considered in this algorithm, an invocation to ***recompile(Program)*** must be performed (Line 15). In this invocation, all loops will be unrolled (Algorithm 7 is invoked inside the compiler), justifying the quadratic complexity. Although our approach is simple, complex heuristics that try to balance code expansion and WCET as proposed by (LOKUCIEJEWSKI; MARWEDEL, 2010) can be applied as well. Our combination of loop unrolling strategies can increase the compilation time due to the fact that we need to process more code that in the original program. Choosing adequate unrolling factors can also increase considerably the compilation time due to the necessity of WCET analyses. As pointed by (LOKUCIEJEWSKI; MARWEDEL, 2010), performance improvement is a primary focus of embedded systems, being longer compilation times of less importance.

6.5 EVALUATION

To evaluate the technique proposed in this chapter we conducted experiments using the infrastructure described in Chapter 5. In the next subsections, we will give a short description of the used resources of the target architecture and implementation aspects. The the last but one section of this chapter presents numerical results obtained from the use of the proposed technique applied to a set of benchmarks commonly used in the literature.

6.5.1 Implementation aspects

As processor resource, we used the extension that enables code predication through the 30th bit, as described in Subsection 5.1 of Chapter 5. The unrolling technique was implemented in our back-end at the end of machine code generation. It is difficult to perform WCET-oriented optimization using LLVM due to its highly optimized pass-manager that isolates the treatment of each function of a compilation unit. Due to this fact, we cannot optimize the program as a whole aiming at WCET reduction using the standard LLVM pass-manager because the generated code is only fully materialized at the end of the complete process. To overcome this situation, we implemented our technique using the approach described in Subsection 5.4.2 of Chapter 5. In this way, the planning tool uses the heuristics of Subsection 6.4.3 to update the database shared with the compiler. The planning tool chooses the loops and unrolling factors and invokes compiler repeatedly until WCET stabilization or when the entire code is already analyzed by the planning tool. From the first compilation, the planning tool can invoke the WCET analyzer tool and execute Algorithm 8 to choose an unrolling factor for each loop.

6.6 RESULTS

We used the Mälardalen WCET benchmarks (GUSTAFSSON; BETTS, 2010) to evaluate the effectiveness of the proposed technique. These benchmarks are widely used to evaluate and compare methods and techniques related to WCET analysis. We excluded benchmarks with indirect recursion. We considered a constant time for complex library function calls, as those that are used to handle floating point numbers. The description of each benchmark is shown in Table 8.

The results of the experiments are shown in Table 9. The column *Initial WCET* shows the WCET of the benchmark without the application of the loop unrolling. *Initial code size* presents the size of the code (in bytes) in this initial scenario. *Optimized WCET* presents the WCET of the optimized version with its respective code size (*Optimized code size*). The columns *WCET reduction* and *Code increase* present the percentage of WCET reduction and its relative code augmentation, respectively. *WCET reduction* is calculated as $\frac{\text{Initial WCET}-\text{Optimized WCET}}{\text{Initial WCET}} \times 100$ and *Code increase* as $\frac{\text{Optimized code size}-\text{Initial code size}}{\text{Initial code size}} \times 100$. We omitted in this table benchmarks where no gain was obtained. In this way, a total of 18 from 33 benchmarks are shown.

Table 10 shows how many loops of each type were unrolled and the maximum unrolling factor (*Max. uf*) in each benchmark.

Analyzing the obtained results, we can see that the combination of techniques was able to reduce the WCET of half of the benchmarks. For example, considering the *adpcm.c* benchmark, we achieved a small WCET reduction (1.19%) in contrast with a higher code increase (31.10%). If we look at Table 10 we can see that two loops of *adpcm.c* were unrolled (one with fixed execution count and another with a call instruction), and the maximum unrolling factor used was 2. On the other hand, we can see a high WCET reduction for the *exptint.c* benchmark, with less code increase then in the *adpcm.c*. In this benchmark, only one loop was unrolled, with an unrolling factor of 7. The average WCET reduction considering all benchmarks was 6.72%, while the average code increase was 15.56%. As maximum values, we

Table 8 – Used benchmarks from (GUSTAFSSON; BETTS, 2010)

Benchmark	**Description**
adpcm	Adaptive pulse code modulation algorithm.
bs	Binary search for the array of 15 integer elements.
bsort100	Bubblesort program.
cnt	Counts non-negative numbers in a matrix.
compress	Data compression program.
cover	Program for testing many paths.
crc	Cyclic redundancy check computation on 40 bytes of data.
duff	Using "Duff's device" from the Jargon file to copy 43 byte array.
edn	Finite Impulse Response (FIR) filter calculations.
expint	Series expansion for computing an exponential integral function.
fac	Calculates the faculty function.
fdct	Fast Discrete Cosine Transform.
fft1	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm.
fibcall	Simple iterative Fibonacci calculation, used to calculate fib(30).
fir	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.
insertsort	Insertion sort on a reversed array of size 10.
janne_complex	Nested loop program.
jfdctint	Discrete-cosine transformation on a 8x8 pixel block.
lcdnum	Read ten values, output half to LCD.
lms	LMS adaptive signal enhancement. The input signal is a sine wave with added white noise.
ludcmp	LU decomposition algorithm.
matmult	Matrix multiplication of two 20x20 matrices.
minver	Inversion of floating point matrix.
ndes	Complex embedded code.
ns	Search in a multi-dimensional array.
nsichneu	Simulate an extended Petri Net.
prime	Calculates whether numbers are prime.
qsort-exam	Non-recursive version of quick sort algorithm.
qurt	Root computation of quadratic equations.
select	A function to select the Nth largest number in a floating point array.
st	Statistics program.
statemate	Automatically generated code.
ud	Calculation of matrixes.

got 32.44% and 80.19%, for WCET reduction and code increase, respectively.

Our approach could be applied to 7 benchmarks, which are *compress.c*, *duff.c*, *edn.c*, *fft1.c*, *fir.c*, *insertsort.c* and *lms.c*. To understand how much WCET reduction we can achieve with the predicated loop unrolling, we unrolled those three benchmarks with the *branchedLoopUnrolling* instead of *predicatedLoopUnrolling* algorithm, because every loop that can be unrolled with the last method can be unrolled with the first as well. The results are shown in Table 11 with WCET reduction percentages highlighted in Figure 36. We can observe that the predicated loop unrolling has noticeable effects considering the *duff.c*, *fir.c* and *insertsort.c* benchmarks. For *insertsort.c* and *fir.c* benchmarks, using the branched approach, we simply do not achieve WCET reduction, so the loop is kept rolled, which also explains the difference in code sizes. As we can see, the predicated approach, even with its limited applicability, can exploit cases where the standard approach fails to get WCET reduction. In the *edn.c* case, we achieved WCET reduction with negative code decrease because the algorithm could use a higher unrolling factor with the predicated version (9 instead 5 for a branch).

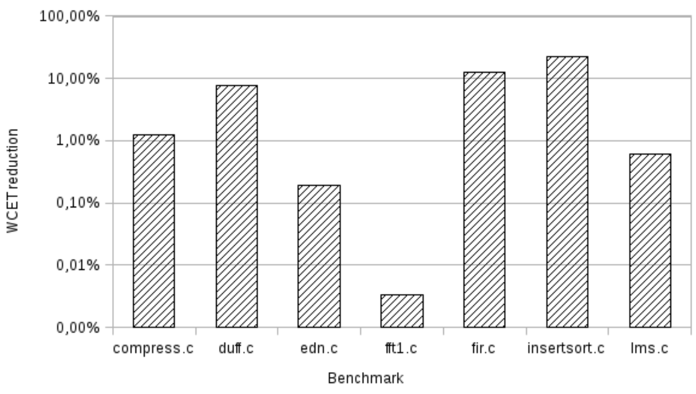


Figure 36 – Graphic comparison of WCET reduction of Table 11.

It is important to say that these results can be enhanced using heuristics to find better unrolling factors to control code expansion (LOKUCIEJEWSKI; MARWEDEL, 2010), which is out of the scope of this work.

6.7 CONCLUSION

Loop unrolling for WCET reduction is considered by (ZHAO et al., 2006) and (LOKUCIEJEWSKI; MARWEDEL, 2010). Though, in both works only loops with fixed iteration counts are unrolled. We proposed in this chapter an alternative way to perform loop unrolling with arbitrary iteration counts. Traditionally, this type of loop is unrolled using compare and branch operations to control different exit conditions or contexts. What we propose is the use of code predication to control the loop execution under different exit conditions, since worst-case analyzers tend to consider that each loop, even unrolled, is always fully executed up to its execution bound. The approach can be used in architectures with full predication support and is best applicable when branch operations are segmented in more than one step.

We introduced an algorithm that performs this code transformation directly at the machine code level (or assembly). In our framework, each data dependent loop of each benchmark is annotated with a safe loop bound that represents an upper bound on the execution count. After loop unrolling application, the annotation is transformed to reflect the new loop bound of the unrolled loop. Since our technique does not depend on branches, the number of instructions is reduced and the instruction scheduling scope is enhanced, as the whole body of the unrolled fits in a single basic block. This scope enhancement can enable more optimizations to be applied to the code.

We also proposed a strategy that selects which unrolling technique to apply in a per loop basis. For loops with fixed execution counts, we applied the standard technique that unrolls loops using unrolling factors that perfectly divide execution counts to avoid compare and branch instructions. For data dependent loops, we used our predicated or the branch-based approach, depending on the case. The approach described here was published in (CARMINATI et al., 2017).

We observed in the experiments that the combination of unrolling techniques was able to reduce the WCET of 18 from 33 benchmarks. For six benchmarks we obtained gains above 20%. In the experiments, we also showed that the predicated approach, even with its limited applicability, can exploit cases where the standard approach fails to get WCET reduction.

As we are not interested in code increase limitation, higher code expansion was observed as well. To work around this situation, techniques like (LOKUCIEJEWSKI; MARWEDEL, 2010) can be applied to our heuristic of unrolling factor selection.

Table 9 – Obtained results

Benchmark	**Initial WCET**	**Initial code size**	**Optimized WCET**	**Optimized code size**	**WCET reduction**	**Code increase**
adpcm.c	19607	10208	19373	14816	1.19%	31.10%
bsort100.c	272623	432	271985	560	0.23%	22.86%
cnt.c	9046	752	8566	864	5.31%	12.96%
compress.c	140139	4912	137162	5488	2.12%	10.50%
crc.c	113846	2048	112671	2624	1.03%	21.95%
duff.c	1859	592	1397	816	24.85%	27.45%
edn.c	96871	2336	73042	3296	24.60%	29.13%
expint.c	113473	1540	76661	1812	32.44%	15.01%
fft1.c	1034634	19984	727844	44272	29.65%	54.86%
fir.c	509930221	976	444387758	1616	12.85%	39.60%
insertsort.c	2720	304	2111	432	22.39%	29.63%
jfdctint.c	3947	1264	3568	1536	9.60%	17.71%
lms.c	352360015	14016	303674957	70768	13.82%	80.19%
ludcmp.c	43902	3296	43622	4320	0.64%	23.70%
matmult.c	268362	1008	225657	1968	15.91%	48.78%
ndes.c	146612	5376	144282	7248	1.59%	25.83%
qsort-exam.c	503582	2528	480816	2784	4.52%	9.20%
st.c	1480353	5088	1198582	5856	19.03%	13.11%
				Average	6.72%	15.56%
				Maximum	32.44%	80.19%

Table 10 – Obtained results

Benchmark	Simple	With pred.	With branch	Max. uf
adpcm.c	2	0	2	2
bsort100.c	1	0	0	5
cnt.c	0	0	1	2
compress.c	1	1	0	2
crc.c	0	0	1	2
duff.c	0	1	0	10
edn.c	4	1	0	9
expint.c	0	1	0	7
fft1.c	0	1	2	13
fir.c	0	1	0	6
insertsort.c	0	1	0	3
jfdctint.c	1	0	0	4
lms.c	0	1	3	15
ludcmp.c	0	0	1	5
matmult.c	1	0	1	4
ndes.c	2	0	1	2
qsort-exam.c	0	0	1	2
st.c	0	0	1	4

Table 11 – Comparing *predicatedLoopUnrolling* with *branchedLoopUnrolling*

Benchmark	**Branched unroll. WCET**	**Branched code size**	**Predicated unroll. WCET**	**Predicated code size**	**WCET reduction**	**Code size reduction**
compress.c	138875	5968	137162	5488	1.23%	8.04%
duff.c	1515	912	1397	816	7.79%	10.53%
edn.c	73181	3072	73042	3296	0.19%	-7.29%
fft1.c	727868	44272	727844	44272	0.00%	0.00%
fir.c	509930221	976	444387758	1616	12.85%	-65.57%
insertsort.c	2720	304	2111	432	22.39%	-42.11%
lms.c	305531002	83248	303674957	70768	0.61%	14.99%

7 CONTRIBUTION 2: ON THE USE OF STATIC BRANCH PREDICTION

The goal of this chapter is to show that a very small or even no gain can be obtained with new optimization techniques targeted to worst-case execution time (WCET) reduction using static predictors. To achieve this, we compare several techniques against the perfect branch predictor. This predictor can estimate the maximum WCET reduction considering static approaches. The comparison includes a new WCET-centered technique which acts as a brute force approach to bring the results as close as possible to the perfect predictor. The comparison also includes standard compiler techniques. As result, we show that all compared techniques are close to the optimal result. We also show that our technique produces slightly better results and WCET-unaware techniques can also be used in real-time environments.

7.1 INTRODUCTION

Most commercial processors available today have resources that increase performance but impose additional difficulties on WCET estimation. A performance enhancement feature can be really problematic for WCET estimation if it causes timing anomalies or domino effects (AXER et al., 2014), as dynamic branch predictors and out-of-order pipelines. Though, real-time applications demand more and more hardware performance, as any computer application. This increasing performance demand pushes the development or the use of deterministic performance enhancement features in real-time systems as well.

A common performance enhancement feature present in advanced modern processors is the so called dynamic branch predictor. Dynamic branch predictors usually have high precision, but they depend on the execution history of the program. This type of predictor increases the average performance but can cause timing anomalies when associated with other performance enhancement features, like instruction caches, as stated by (AXER et al., 2014). Although there are works related to dynamic predictors and WCET (COLIN; PUAUT, 2000) (BATE;

REUTEMANN, 2004) (BURGUIERE; ROCHANGE, 2007) (LOUISE, 2011), its support is not a common feature present in WCET analyzers (WILHELM et al., 2008).

An alternative to dynamic branch predictors is the static branch predictor. These predictors may depend on the compiler to define the behavior of each conditional branch. This behavior is then adopted by the processor for the entire program execution. This strategy for branch prediction can be easily analyzable by WCET tools, although it has inferior average case performance in contrast with the dynamic solution. A usual approach in static branch prediction is to tune the application (or its branches) using profile data (FISHER; FREUDENBERGER, 1992) or static program analysis (PATTERSON, 1995). Regardless of that, the focus of both strategies is to improve the average case performance, which sometimes contradicts real-time performance. We can cite PowerPC 7451 and the MIPS R4000 family as examples of processors with this type of predictor. Simpler static branch predictors that do not depend on the compiler exist as well, like those that are based on the target address (backward taken, forward not taken for example). Infineon TriCore processor TC1796 has this type of branch predictor. Several simple static branch prediction schemes and their accuracy are discussed in (SMITH, 1981) considering execution performance.

The use of static branch prediction as a mechanism for worst-case execution time reduction is a well-known alternative and was firstly proposed by (BODIN; PUAUT, 2005). The proposed scheme is an iterative algorithm that works on the program control flow graph. Another work that proposes reducing WCET using static branch predictors and a compiler is (BURGUIERE et al., 2005). One secondary result presented by (BURGUIERE et al., 2005) is that a compiler directed static branch prediction scheme can improve the WCET more than a dynamic one, considering predictors without aliasing of branches. Another way to reduce WCET based on static branch prediction is by changing the code layout of a program, as done in (PLAZAR et al., 2011). Though, this technique is applied to predictors that can not be tuned by the compiler. The technique of (BODIN; PUAUT, 2005), is explained in details in Chapter 4, more specifically in the assembly level optimizations subsection. The work of (BURGUIERE et al., 2005) is also explained in Chapter 4.

The contributions of our work are:

- To show that the use and development of new techniques aiming at WCET reduction using static branch prediction techniques reached the limit in terms of worst-case execution time improvement. Regardless of the approach employed to predict the behavior of each branch of a program, the result will always be close to an optimal result. Though this is the main objective of our work, it will be achieved as a corollary result of the two next contributions.

- To obtain an optimal WCET in terms of static branch prediction we use the concept of *perfect branch predictor* (COLIN; PUAUT, 2000). This is a virtual branch predictor that can be easily analyzable by a WCET analyzer, but can not be implemented in practice. This predictor can give an asymptotic limit for the gain in terms of WCET reduction, considering static prediction schemes. We also used this predictor to calculate the effective zone of actuation of static prediction techniques. This zone is a WCET interval where, for every program, the maximum value considers that every branch will be mispredicted and minimum value considers a correct prediction of all branches.

- We compare several prediction schemes against the *perfect branch predictor*. Approaches that do not depend on WCET calculation are also considered, because they are a cheaper alternative than the WCET-centered ones. As an example of such approach, we can cite the use of standard compiler information to predict branches. At the time of writing this chapter, no work exists comparing the worst-case performance of several static prediction schemes, with the purpose of exposing trade-offs of usability concerning real-time systems. We also include in this comparison a new technique that explores a situation where the classic approach makes sub-optimal choices. This approach does not depend on specific changes considering WCET analyzers, and it acts as a brute force approach to bring the results as close as possible to the perfect predictor (or the optimal result).

The remainder of this chapter is organized as follows. Section 7.2 revises the perfect predictor approach. Section 7.3 presents our WCET branch prediction approach, and Section 7.4 presents how we evaluated all considered approaches against the perfect predictor and shows the obtained results. In Section 7.5, we present our conclusions.

7.2 THE PERFECT BRANCH PREDICTOR APPROACH

A *perfect branch predictor* (COLIN; PUAUT, 2000) is a conceptual predictor that knows the correct target of each branch before the execution, which means that it never misses a branch prediction. This predictor does not suffer from misprediction penalties, which does not imply the non-existence of control flow transfer penalties. Although not implementable, this predictor behavior can be easily included in a WCET analyzer. Assuming that we can calculate the WCET of a program using the perfect predictor, it is possible to estimate an asymptotic limit for the WCET of any static branch prediction configuration of this program. We consider this limit as asymptotic because it refers to hardware states that can not be reached in practice. For a program P, we can define some possible WCET values considering different prediction schemes:

- $WCET_{all-mispredicted}(P)$: these values represent the WCET considering that all branches will be mispredicted, i.e., penalization is considered for taken and fall-through targets. This scenario represents the opposite behavior of the *perfect* predictor.

- $WCET_{perfect}(P)$: these values represent a situation when the perfect branch predictor is used.

- $WCET_X(P)$: finally, these values represent the WCET for some static branch prediction scheme X.

We can also define a relationship between the previous values such that: $WCET_{all-mispredicted}(P) \geq WCET_X(P) \geq WCET_{perfect}(P)$ for any X and P. In this way, we can verify the effectiveness of any WCET reduction scheme X applied to a program P measuring the distance between the values of $WCET_X(P)$ and $WCET_{perfect}(P)$. It is important to say that this inequalities are valid only for architectures without timing anomalies. If we assume that for every program P, a $WCET_{perfect}(P)$ represents the optimal WCET, we can can define the effective zone of actuation between the best WCET and the worst possible WCET:

$$[WCET_{perfect}(P), WCET_{all-mispredicted}(P)] \tag{7.1}$$

Using interval from Equation 7.1, we can calculate the percentage of optimality of a prediction scheme over a program P. If $WCET_X(P) = WCET_{all-mispredicted}(P)$ for a prediction scheme X, we say that this prediction scheme is 0% optimal. On the other hand, if we have $WCET_X(P) = WCET_{perfect}(P)$ we call this prediction scheme 100% optimal. For intermediate cases, we can calculate the optimality with the following equation (Equation 7.2):

$$\frac{WCET_{all-mispredicted}(P) - WCET_X(P)}{WCET_{all-mispredicted}(P) - WCET_{perfect}(P)} \times 100 \tag{7.2}$$

Equation 7.2 will be used to compare the overall performance of all used prediction schemes in the evaluation section.

From the WCET analysis perspective, we can obtain a perfect behavior by removing all penalties attributed to the execution of branch operations. Unconditional control flow transfer penalties shouldn't be removed because their execution are not effected by branch prediction units. Usually, branch overhead is modeled as edge weight (from one basic block to another) in the CFG of a program under WCET analysis, or directly in the basic block times. If this is the case, we can reduce these weights/times to reflect the always well-predicted behavior.

This behavior can not be implemented in practice because it needs a clairvoyant BTB (*branch target buffer*) and a processor that is capable of executing speculatively and simultaneously both fall-through and taken successors of each branch until the condition resolution.

7.3 A NEW TECHNIQUE TO REDUCE WCET USING BRANCH PREDICTION

Before the presentation of our new technique, we will discuss some aspects of the classic approach (Algorithm 1).

7.3.1 Considerations on the classic approach

Considering commercial WCET analyzers, we can not change them to mispredict a specific branch in both directions on demand, then, we must consider the behavior of branch instructions when executed in hardware. In this case, the penalty is only considered in the case of a wrong prediction. A wrong prediction occurs when the processor follows a different branch target from that indicated by the compiler. Furthermore, for some processors, only commercial analyzers are available and we can not change them to support misprediction penalties for both targets of a branch.

The advantage of using an unchanged WCET analyzer is that we can produce better results when conducting any WCET-aware code optimization. With more precise WCET estimates (considering caches, for example), we can consider more realistic cases of WCEP changes. In this work, we apply the techniques to a deterministic processor without timing anomalies. But, if these techniques are applied to an architecture with timing anomalies such as PowerPC 755, different predictions of a branch can lead to different timing behaviors of instructions that are executed after that branch, affecting the WCET of the path in a non-obvious way. These anomalies are related to the instruction cache and are called *speculation-caused anomalies*, as showed in (WILHELM et al., 2008).

If we consider an unchanged WCET analyzer, branch prediction guided by WCET information of a program can lead to WCET increase if we use Algorithm 1. To illustrate this situation, let's consider the example shown by Figure 37. It represents the CFG of a hypothetical program that is basically an if-then-else structure.

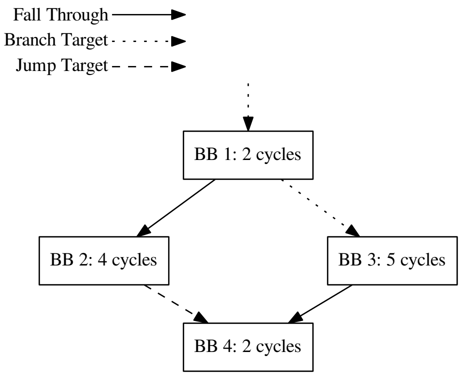


Figure 37 – Example of an if-then-else sentence.

In processors with static branch predictors, we must choose a static branch direction for each branch. If we do not know the common direction of each branch, a default one must be used. The most common approach is to use the *not taken* direction (fall-through path) as default, because this choice results in less penalty in the case of a misprediction. To explain how misprediction penalties occur, we may consider the decision tree of Figure 38. If a branch is predicted as not taken and the condition is evaluated to false, no penalty occurs, because fall-through path, which is the correct one, has already been taken by the processor pipeline. However, if this same condition is evaluated to true, the fall-through path must be flushed from the pipeline and the program flow must be resumed along the correct path (branch target).

On the other hand, if a branch is predicted as *taken*, and the branch target address is determined, its path is executed until the res-

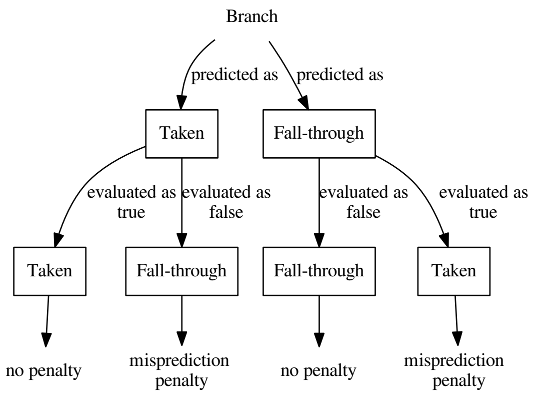


Figure 38 – Decision tree that describes when penalties occur (or not) in the case of static branch prediction.

olution of the condition. If the condition is evaluated to true, we possibly have only the overhead of the cycles needed for the address calculation, that is generally executed some cycles after the decode stage of a pipelined processor. Such overhead appears as a sequence of pipeline stalls between the branch instruction and the instruction located at the taken address. But, if the condition is evaluated to false, the taken path must be flushed and the execution must be resumed along the fall-through path.

Considering a default *not taken* approach, we can calculate the WCET of the example illustrated by Figure 37. So we have two possible execution paths for this example:

- BB 1 $\rightarrow$ BB 2 $\rightarrow$ BB 4: The time of this path is the sum of the basic-block times added with the penalization of a jump (or unconditional branch) between basic blocks BB 2 and BB 4. There is no penalty between basic blocks BB 1 and BB 2, because BB 2 is in the fall-through path of BB 1, and *not taken* is the prediction. So, the time of this path will be $2+4+2+\mathit{jumpPenalty} =$

$8 + jumpPenalty$.

- BB 1 $\rightarrow$ BB 3 $\rightarrow$ BB 4: The time of this path is the sum of the basic-block times added with the penalization of a mispredicted branch between basic blocks BB 1 and BB 3 and a jump penalization between basic blocks BB 3 and BB 4. So, the time of this path will be $2+5+2+mispredictionPenalty_{notTaken}+jumpPenalty$, which results in $9+mispredictionPenalty_{notTaken}+jumpPenalty$.

Comparing the previous paths, one can see that the worst-case path is BB 1 $\rightarrow$ BB 3 $\rightarrow$ BB 4. If we apply the WCET-oriented static branch prediction scheme in this example (misprediction penalty for only one successor), we will set the direction of the branch present in BB 1 as *taken*, to reduce the penalty from transferring the control from BB 1 to BB 3. Doing this, we will reduce the time of this path (factor $mispredictionPenalty_{notTaken}$). Although, by predicting this branch as *taken*, we will impose a penalization over the fall-through path, when the condition is evaluated to false. Now the time of the path BB 1 $\rightarrow$ BB 2 $\rightarrow$ BB 4 is augmented by one branch penalization and becomes $8+mispredictionPenalty_{taken}+jumpPenalty$. In this scenario, a WCEP switch can occur, followed by a WCET increase. This WCEP switch can occur because a misprediction penalty of a branch originally predicted as taken is higher than an originally predicted as not taken ($mispredictionPenalty_{taken} > mispredictionPenalty_{notTaken}$).

To illustrate another aspect of the classic approach, we can consider the code snippet of Code 7.1. This code relates to a sequence of three loops, and its control-flow graph is presented by Figure 39. This situation occurs when the compiler optimizes the initialization of loop counters. For example, Clang/LLVM may generate code in this way depending on the optimization level used or the optimizations carried in the target code generator.

If we apply Algorithm 1 to this example it will encounter the following situation for the basic block related to the first loop header (*header 0* of Figure 39) and its successors:

Code 7.1 – Code snippet relative to a sequence of three loops

```
int i, j, k;

/*Loop 0*/
for (i = 0; i < 10; i++) {
    /*...*/
}
/*Loop 1*/
for (j = 0; j < 10; j++) {
    /*...*/
}
/*Loop 2*/
for (k = 0; k < 10; k++) {
    /*...*/
}
```

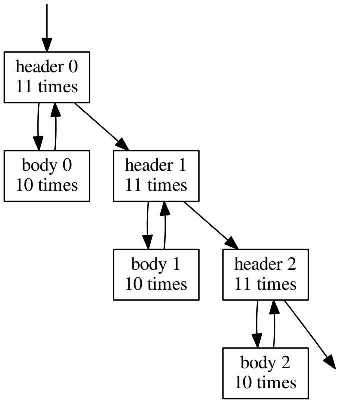
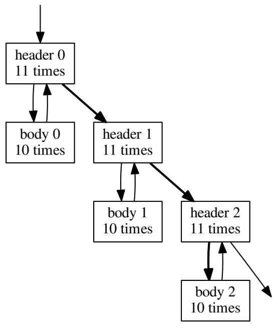


Figure 39 – Control flow graph of Code 7.1.

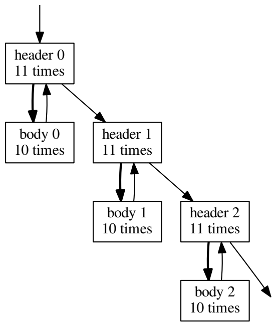
- Both successors *header 1* and *body 1* are on the WCEP. In this situation, Line 13 of Algorithm 1 will be evaluated to true.

- The control transfer to the successor with the highest count (in this case, *header 1*) must be predicted (Line 14 of Algorithm 1) to reduce the control flow penalization. Depending on how the code is generated, loop headers always execute once more on the last iteration when the exit branch is taken.

However, this prediction will highly penalize the execution of the first loop, because the flow transfer from *header 0* to *body 0* always will be mispredicted, despite of the *header 0* to *body 0* transfer be executed only once. This situation is repeated for Loop 1. For Loop 2 this situation does not occur because Header 1 does not have a successor with a count higher than the basic block *Body 2*. The final prediction (in bold edges) is showed in Subfigure 40a. This figure shows that Loop 0 and 1 will be penalized by the chosen predictions. For comparison purposes, Subfigure 40b shows an optimal prediction, that optimize the execution of all loops.



(a) Branches predicted by Algorithm 1.



(b) Optimal prediction of branches.

Figure 40 – Example of predictions of the branches of Figure 39. Edges in bold represent the predicted target of the branch.

7.3.2 The proposed technique

This subsection describes the technique proposed in this chapter, its motivation and computational complexity. One motivation behind the algorithm is that, if we use an unchanged WCET analyzer for a processor with static branch prediction for complete programs, the assumption that a branch is not predicted for both *taken* and *fall-through* successors can not be used, and that is necessary for Algorithm 1.

Our approach for static prediction of branches consists in iteratively choosing predictions along the WCEP of the program, like the classic approach. Differently from the classic approach, our strategy does not assume that a misprediction will occur in both *taken* and *fall-through*. We only consider that there is a penalization to execute successors that are different from the successor executed by default by the processor. In this way, if a processor executes by default its fall-through successor on branch operations, we will consider the penalization only when the execution path follows the *taken* successor. Doing so we can use this technique in conjunction with any WCET analyzer that supports processors with static branch prediction. The technique is showed in Algorithm 9.

The first task of the algorithm is to mark all basic blocks as unpredicted, and set a default direction as *fall-through*, which is done by Lines 4 to 9. The first iterative step of the algorithm (*Step 1*) is to calculate the WCET of the program (Line 11). The procedure *estimate_WCET* returns an object that contains both the WCEP and its total execution time (WCET) in processor cycles. The entire interface with a WCET analyzer is encapsulated by this procedure. *Step 2* consists of predicting the behavior of branches while there are WCEP changes. For this purpose, the algorithm iterates over all basic blocks in the WCEP and:

- If a basic block is marked as predicted, it will be ignored. Otherwise, it will be marked as predicted (Lines 16 and 17) and the algorithm executes the next stage.

Algorithm 9 Proposed approach for static branch prediction.

```
1: procedure SET_PREDICTIONS($CFG$) ▷ Set predictions along the WCEP
2:     bool $converged \leftarrow false$
3:     bool $wcet\_changed \leftarrow false$
4:     for all $BB \in CFG$ do
5:         $BB.predicted \leftarrow false$
6:         if $is\_conditional\_branch(BB)$ then
7:             $BB.direction \leftarrow fall-through$
8:         end if
9:     end for
10: {Step 1: WCET estimation}
11:     $WC \leftarrow estimate\_WCET(CFG)$
12:     while $converged = false$ do
13: {Step 2: Issue static branch predictions along the WCEP}
14:         $wcet\_changed \leftarrow false$
15:         for $BB \in WC.WCEP \wedge wcet\_changed = false$ do
16:             if $BB.predicted = false$ then
17:                 $BB.predicted \leftarrow true$
18:                 if $is\_conditional\_branch(BB)$ then
19:                     if $count(BB.tk) > count(BB.ft)$ then
20:                         $BB.direction \leftarrow taken$
21:                         $new\_WC \leftarrow estimate\_WCET(CFG)$
22:                     {Step 3: Verify any WCET increase}
23:                         if $new\_WC.WCET > WC.WCET$ then
24:                             $BB.direction \leftarrow fall-through$
25:                         else
26:                             $WC \leftarrow new\_WC$
27:                             $wcet\_changed \leftarrow true$
28:                         end if
29:                     end if
30:                 end if
31:             end if
32:         end for
33:         if $\forall BB \in WC.WCEP, \quad BB.predicted = true$ then
34:             $converged \leftarrow true$
35:         end if
36:     end while
37: end procedure
```

- If a basic block ends with a conditional branch (Line 18, which is given by function *is_conditional_branch*) we predict this branch as *taken*, if the *taken* successor executes more than the *fall-through* successor (Lines 19 and 20). If this is the case, we must recalculate the WCET and verify if it was increased (Lines 21 and 23) which is called *Step 3*. If a WCET increase is detected, then we must return the prediction to fall-through and proceed to the next basic block. If the WCET has not increased, we must update the current WCET information and restart the iteration over the new WCEP (Lines 26 and 27).

The algorithm ends with WCET convergence, i.e., when all basic blocks in the current WCET are marked as predicted. This algorithm is intended to be a generalization of the classic algorithm. In this way, it prevents the WCET increase when we have no control on how the WCET is calculated. We can summarize the differences of the technique proposed here and the classic approach:

- Analyzer requirement: Algorithm 1 needs a change of the WCET analyzer to support misprediction penalties at every branch direction. In this way, each execution path is penalized *a priori* of the execution of the algorithm. Our algorithm does not require a tailored analyzer. Our algorithm only requires a worst-case execution count for each basic block.

- WCET recalculation due to WCEP switches: Algorithm 1 recalculates the WCET at the end of path traversal. As we do not use an all-mispredicted WCET analyzer feature, each static branch prediction can lead to a WCET increase and a WCEP switch. Testing WCET increases and WCEP switches on the fly permit us to explore more possibilities of prediction.

The complexity of the algorithm is proportional to the number of branches that exist in a program. In the worst case, all n conditional branches that appear in the program will also appear in the WCEP. In this case, there will be n WCET calculations, because the algorithm tries to predict each branch as taken exactly once. This algorithm has a higher overhead when compared with the the classic approach, because it calculates the WCET after an entire WCEP traversal. As stated by (LOKUCIEJEWSKI; MARWEDEL, 2011), compilation overhead is acceptable for embedded real-time system, where WCET reduction is important to enhance the schedulability of the system, or even reduce resource demand.

In the next section we will explain the concept of perfect branch predictor, as a tool to evaluate WCET reduction schemes related to static branch prediction.

7.4 EVALUATION OF TECHNIQUES AGAINST THE PERFECT PREDICTOR

To evaluate the techniques, we conducted experiments using using the infrastructure described in Chapter 5. We also included in the comparison a compiler heuristic and a simple "not taken" approach. The compiler heuristic used is a standard compiler feature and will be discussed further.

As processor resource, we used the extension that enables static branch prediction through a special instruction that allows the compiler to indicate whether a given path is more likely to be executed. This resource is described in Subsection 5.1 of Chapter 5.

Both techniques were implemented in our back-end at the end of machine code generation. It is difficult to perform WCET-oriented optimization using LLVM due to its highly optimized pass-manager that isolates the treatment of each function of a compilation unit. To overcome this situation, we implemented our technique using the approach described in Subsection 5.4.1 of Chapter 5. Such approach allows us to make any change to any function and regenerate the object code file, at the end of all LLVM code generation passes. With this LLVM adaptation, any change at the machine code level can be done in any function, followed by an entire object code generation, many times as needed.

In relation to the implementation of the static prediction schemes inside the compiler, a free slot is always reserved in a bundle to hold a *preld* instruction for each branch instruction found in a program. If a branch is predicted a *taken*, a *preld* instruction is generated in such slot. Hence, if a branch is predicted as *not taken*, a *nop* is generated instead. We do this to generate binaries with the same number of instructions, no matter the prediction scheme used. Different number of instructions can produce different cache timing behavior due to the code alignment, for example. In this way, we can execute experiments and only observe timing variations caused by the prediction influence on the WCET. At the linking stage, we use *scratchpad memory* (SPM) to hold all program data and stack, except instructions that are stored in an instruction memory. Although scratchpad allocation strategies could be employed as well, it is not the objective of this chapter.

7.4.1 Results

We used a set of examples from the Mälardalen WCET benchmarks (GUSTAFSSON; BETTS, 2010) to evaluate the effectiveness of the techniques. These benchmarks are commonly used to evaluate and compare methods and techniques related to WCET analysis. We excluded benchmarks which use library functions that are directly or indirectly called by the generated code, floating-point calculations and indirect recursion. The description of each benchmark is shown in Table 12. The restriction of absence of library function calls exists because we have a very simple runtime support (including boot loader) that is linked with the application. This runtime does not implement library support, therefore the application must be self-contained.

Table 13 shows the WCET values for all used benchmarks considering *all mispredicted* and *perfect* approaches. It also shows the improvement of the *perfect* over the *all mispredicted* scheme. This is the asymptotic limit for any static branch prediction scheme that aims at WCET reduction. The effective zone size for each benchmark is also shown in this table. This size represents the term $WCET_{all-mispredicted}(P)$ - $WCET_{perfect}(P)$ from Equation 7.2, for each benchmark P.

The results of the experiments with the other methods are shown in Table 14. This table shows the results in clock cycles for each benchmark, considering five cases:

- Not taken: In this case, all branches are predicted as *not taken*.
- WCEP-aware: This scenario shows the application of Algorithm

Table 12 – Used benchmarks from (GUSTAFSSON; BETTS, 2010).

Benchmark	Description
adpcm.c	Adaptive pulse code modulation algorithm.
bs.c	Binary search for the array of 15 integer elements.
bsort100.c	Bubblesort program.
cnt.c	Counts non-negative numbers in a matrix.
cover.c	Program for testing many paths.
crc.c	Cyclic redundancy check computation on 40 bytes of data.
duff.c	Using "Duff's device" from the Jargon file to copy 43 byte array.
edn.c	Finite Impulse Response (FIR) filter calculations.
fac.c	Calculates the faculty function.
fdct.c	Fast Discrete Cosine Transform.
fibcall.c	Simple iterative Fibonacci calculation, used to calculate fib(30).
insertsort.c	Insertion sort on a reversed array of size 10.
janne_complex.c	Nested loop program.
jfdctint.c	Discrete-cosine transformation on a 8x8 pixel block.
lcdnum.c	Read ten values, output half to LCD.
matmult.c	Matrix multiplication of two 20x20 matrices.
ndes.c	Complex embedded code.
ns.c	Search in a multi-dimensional array.
nsichneu.c	Simulate an extended Petri Net.
prime.c	Calculates whether numbers are prime.

1 that is related to the classic approach. In this scenario, the WCET calculation uses a 2-way misprediction penalty for non-predicted branches, as required by the algorithm. We call this algorithm WCEP-aware because it only relies on the WCEP of a program. We call this misprediction penalty as 2-way because it is considered for both fall-through and taken successors.

- WCET-aware: This scenario shows the application of the technique proposed in this chapter (Algorithm 9), which uses the standard behavior of the WCET analyzer. We call this algorithm *WCET-aware* because it depends on the numerical values of the WCET, and not only on the WCEP.

- Compiler native heuristic: Some compilers perform analyses that can be used to statically predict the behavior of a branch. Such analyses can, for example, estimate the probability of each branch to be taken by the program using heuristics. For example, both LLVM(LATTNER; ADVE, 2004) and GCC can calculate this in-

Table 13 – Definition of the maximum possible WCET improvement considering the *perfect predictor* for a subset of the Mälardalen benchmarks(GUSTAFSSON; BETTS, 2010). The improvement is calculated using *All mispredicted* as baseline.

Benchmark	**All mis-predicted**	**Perfect**	**Improv.**	**Effectiv. zone size**
adpcm.c	20747	18484	10.91%	2263
bs.c	378	308	18.52%	70
bsort100.c	447176	269259	39.79%	177917
cnt.c	10981	8995	18.09%	1986
cover.c	5760	4125	28.39%	1635
crc.c	164734	112978	31.42%	51756
duff.c	1966	1615	17.85%	351
edn.c	118332	96135	18.76%	22197
fac.c	1560	1318	15.51%	242
fdct.c	1984	1922	3.13%	62
fibcall.c	750	486	35.20%	264
insertsort.c	3161	2591	18.03%	570
janne_complex.c	6333	3684	41.83%	2649
jfdctint.c	4033	3752	6.97%	281
lcdnum.c	1073	860	19.85%	213
matmult.c	323938	266989	17.58%	56949
ndes.c	152390	141446	7.18%	10944
ns.c	29452	22438	23.82%	7014
nsichneu.c	59620	51695	13.29%	7925
prime.c	42625	34037	20.15%	8588

formation examining the structure of the CFG of the program, if an execution profile is not available. This information is obtained through a set of heuristics that act on the structure of loops and its exit conditions. The examination of comparison operations that precede a branch instruction is also a key aspect of these heuristics. In this case, we used branch probabilities obtained from the *Machine Branch Probability Info* pass of LLVM version 3.3, which is an analysis pass available on the LLVM infrastructure. This heuristic can be implemented in any compiler with such analysis available, although, results can vary depending on the obtained probabilities. The objective of this scheme is to assert if non WCET-centric schemes can be used for WCET reduction. This strategy is summarized by Algorithm 2 presented in Subsection 5.2.1 of Chapter 5.

We did not included in the comparison the approach proposed in (BURGUIERE et al., 2005), because that technique needs a better structured program regarding loops than what is generated by our compiler. That technique also needs the ability of calculating the WCET of parts of a program, which is not a feature commonly available in analyzers.

Table 14 also shows the improvement of each technique in relation to the *all mispredicted* column of Table 13.

The results of Table 14 show that all techniques presented similar results. Disregarding tied cases, the best results were obtained by the WCET-aware approach. An interesting result is that the compiler heuristic obtained more WCET reduction than the WCEP-aware strategy for some benchmarks. This effect can be observed in the examples *crc.c*, *jfdctint.c* and *nsichneu.c*, which means that the WCEP-aware was not able to identify the best direction for some branches of the benchmarks. The *Not taken* approach was responsible for the worst result, but still tied in several cases.

Comparing the results against the *perfect branch predictor*, one can see that the best results obtained for each benchmark are close to the WCET calculated considering the perfect case. Figure 41 graphically shows the WCET reduction in relation to the all-mispredicted case for the three approaches plus the reduction obtained with the *perfect* predictor for all benchmarks. We can see in this graph that the results obtained with these approaches are so close to the result obtained with the perfect predictor, that little or none gain can be achieved with the development of new techniques for static prediction of branches aiming at WCET reduction.

To better understand the results, we can use the effective zone (defined in Section 7.2) for each benchmark, and use it to calculate how far from the optimal prediction each approach is. The results are shown in Table 15 and were calculated using Equation 7.2. This table also shows the count of the best results obtained for each technique considering the used benchmark. The obtained results can also be seen in Figure 42, which is a graphic representation of Table 15. As we can see, all approaches exhibited similar optimality percentage. The best result was obtained for the *WCET-aware*, whereas the worst was given by the *Not taken* strategy. The second best result was obtained by the *WCEP-aware* algorithm. Though, the compiler heuristic presented very good results when compared with WCET-centered approaches. This result shows that a simple and fast heuristic can be used as well to reduce the WCET of applications. The *Not taken* approach stays approximately 15% below the other approaches. Table 15 also shows which technique produced the best result for each benchmark. We consider as best result the technique that achieved greater WCET reduction, which is marked as bold in the table. Regarding the overall average case, the technique proposed in this chapter is able to produce better results, when compared with the other techniques.

Analyzing the results of Table 15 we can see that we obtained a distinguished difference between *WCEP-aware* and *WCET-aware* approaches considering the *jfdctint.c* benchmark. The code snippet responsible for this difference is presented in Code 7.2. In function *jpeg_fdct_islow*, we have two loops that have the same number of iterations that is 8. The control flow graph of this piece of code can be seen in Figure 43. If we apply Algorithm 1 to *jdfctint.c*, it will encounter a situation similar to that found in the example from Code 7.1.

This situation is avoided by our approach, because it checks for any WCET increase along the process. If we consider only execution counts, some branches can not have the best prediction that is possible. Figure 43 also shows which flow transfer will be predicted by both techniques.

The time to run a code generation followed by an optimization depends on the technique used, the size of the program and the number of branches. For example, considering small benchmarks such as *fibcall.c*, *bsort100.c* and *prime.c*, it takes around 0.5 seconds on an ordinary desktop computer to execute both our technique and the classic approach. Although, for bigger benchmarks we can note significant differences. For example, our technique took around 41 seconds to process the *adpcm.c* benchmark, whereas the classic approach processed the same code in approximately 4 seconds.

Code 7.2 – Code snippet of *jfdctint.c*, where *DCTSIZE* is 8

```
void jpeg_fdct_islow () {
    /* ... */
    int ctr;
    /* ... */
    for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
        /* ... */
    }
    for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
        /* ... */
    }
}
```

7.5 CONCLUSION

Nowadays processors use performance improvement features that are targeted to the average-case execution time, such as caches, out-of-order execution, and branch predictors. However, some features are incompatible with real-time applications due to WCET related issues. Dynamic branch predictors are an example of such features. They are difficult to model in WCET analysis, due to characteristics like aliasing, on which one branch can interfere with the prediction of another branch. However, static branch prediction is a processor technology that is employed to overcome the analyzability problem in the case of real-time systems. As showed by (BURGUIERE et al., 2005), static branch predictors outperform simple and analyzable dynamic ones in terms of WCET bounds, so this chapter focused on the static type of predictor.

With the use of the *perfect branch predictor* (COLIN; PUAUT, 2000) concept, applied to a subset of the WCET benchmarks adopted by the literature, we showed that a very small or even no gain can be obtained with new techniques targeted to WCET reduction considering static branch prediction. This means that the techniques considered in this chapter are close to an optimal result.

To achieve this conclusion, we evaluated basically four techniques: the classic approach (BODIN; PUAUT, 2005), a new approach proposed in this chapter, a *not taken* approach and one compiler heuristic. We proposed a new technique because the classic approach depends on a change on how the WCET is calculated for a determined program: a fictional branch instruction behavior can be implemented on the analyzer, which generates penalties for both *taken* and *fall-through* destinations, or the integer linear programming constraints which model the program flow must be changed to make room for these simultaneous penalties. If we can tune the WCET analyzer to consider misprediction penalties for both branch targets (*taken* and *fall-through*), one can use the classic approach that is illustrated by Algorithm 1. But, if we can not change the WCET tool, or it implements the strict hardware behavior, one can use the technique proposed in this chapter, which is generalized by Algorithm 9. The handling of loops was another reason to propose a different technique.

As a secondary result, we showed experimentally that our technique improved the WCET of more programs, excluding cases of tie, when more than one technique obtained the same WCET reduction. We know that our algorithm is a brute-force strategy and has high computational complexity of code generation when compared with alternative approaches, but it presented the best results. Though, by using our approach, we can avoid undesirable effects, like that evidenced by *jfdctint.c*, where a flow transfer inside of a loop was penalized. We also obtained very good results using a heuristic that uses information already available in the compiler about the probability of branches. This heuristic can be used if the cost of WCET-centered techniques is not tolerable for application development.

Table 14 – Results of optimizing the subset of benchmarks using five different approaches. The improvement is calculated using *All mispredicted* as baseline (Table 13).

Benchmark	**Not taken**	**Improv.**	**WCEP-aware**	**Improv.**	**WCET-aware**	**Improv.**	**Heur.**	**Improv.**
adpcm.c	19621	5.43%	18990	8.47%	18941	8.70%	19029	8.28%
bs.c	320	15.34%	320	15.34%	320	15.34%	320	15.34%
bsort100.c	270156	39.59%	269567	39.72%	269567	39.72%	269567	39.72%
cnt.c	9061	17.48%	9061	17.48%	9061	17.48%	9061	17.48%
cover.c	4672	18.89%	4146	28.02%	4146	28.02%	4672	18.89%
crc.c	117684	28.56%	116172	29.48%	116156	29.49%	116162	29.49%
duff.c	1918	2.44%	1624	17.40%	1624	17.40%	1624	17.40%
edn.c	97410	17.68%	96528	18.43%	96528	18.43%	96528	18.43%
fac.c	1336	14.36%	1326	15.00%	1326	15.00%	1326	15.00%
fdct.c	1970	0.71%	1936	2.42%	1936	2.42%	1936	2.42%
fibcall.c	657	12.40%	579	22.80%	579	22.80%	579	22.80%
insertsort.c	2621	17.08%	2621	17.08%	2621	17.08%	2621	17.08%
janne_complex.c	3741	40.93%	3741	40.93%	3741	40.93%	3741	40.93%
jfdctint.c	3971	1.54%	3814	5.43%	3769	6.55%	3769	6.55%
lcdnum.c	893	16.78%	893	16.78%	893	16.78%	893	16.78%
matmult.c	268378	17.15%	268378	17.15%	268378	17.15%	268378	17.15%
ndes.c	146616	3.79%	143017	6.15%	143017	6.15%	143065	6.12%
ns.c	24766	15.91%	22900	22.25%	22900	22.25%	22900	22.25%
nsichneu.c	51698	13.29%	51766	13.17%	51698	13.29%	51698	13.29%
prime.c	34049	20.12%	34043	20.13%	34043	20.13%	34049	20.12%

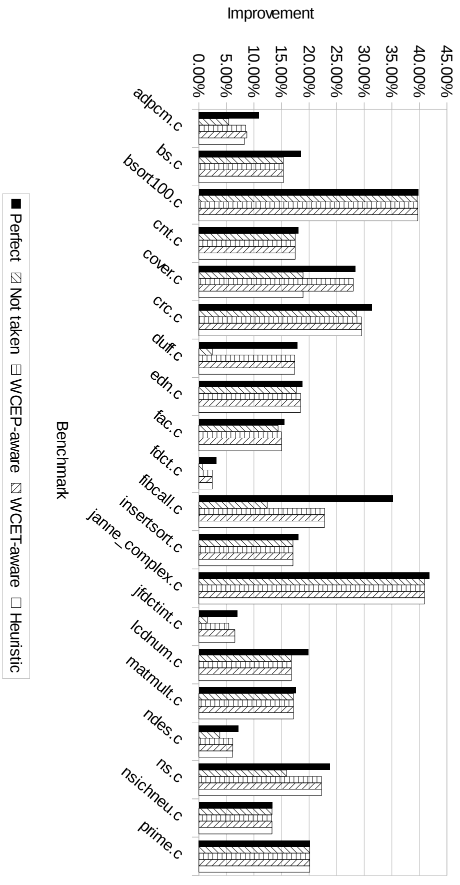


Figure 41 – Bar chart representing the WCET improvement for each technique in relation to the *all-mispredicted* case. The WCET improvement considering the *perfect* predictor is also shown for comparison purposes.

Table 15 – Optimality of the five approaches (100% is the optimum).

Benchmark	Not taken	WCEP-aware	WCET-aware	Heur.
adpcm.c	49.76%	77.64%	**79.81%**	75.92%
bs.c	**82.86%**	**82.86%**	**82.86%**	**82.86%**
bsort100.c	99.50%	**99.83%**	**99.83%**	**99.83%**
cnt.c	**96.68%**	**96.68%**	**96.68%**	**96.68%**
cover.c	66.54%	**98.72%**	**98.72%**	66.54%
crc.c	90.91%	93.83%	**93.86%**	93.85%
duff.c	13.68%	**97.44%**	**97.44%**	**97.44%**
edn.c	94.26%	**98.23%**	**98.23%**	**98.23%**
fac.c	92.56%	**96.69%**	**96.69%**	**96.69%**
fdct.c	22.58%	**77.42%**	77.42%	**77.42%**
fibcall.c	35.23%	**64.77%**	**64.77%**	**64.77%**
insertsort.c	**94.74%**	**94.74%**	**94.74%**	**94.74%**
janne_complex.c	**97.85%**	**97.85%**	**97.85%**	**97.85%**
jfdctint.c	22.06%	77.94%	**93.95%**	**93.95%**
lcdnum.c	**84.51%**	**84.51%**	**84.51%**	**84.51%**
matmult.c	**97.56%**	**97.56%**	**97.56%**	**97.56%**
ndes.c	52.76%	**85.65%**	**85.65%**	85.21%
ns.c	66.81%	**93.41%**	**93.41%**	**93.41%**
nsichneu.c	**99.96%**	99.10%	**99.96%**	**99.96%**
prime.c	99.86%	**99.93%**	**99.93%**	99.86%
Count of best results	7	16	20	15

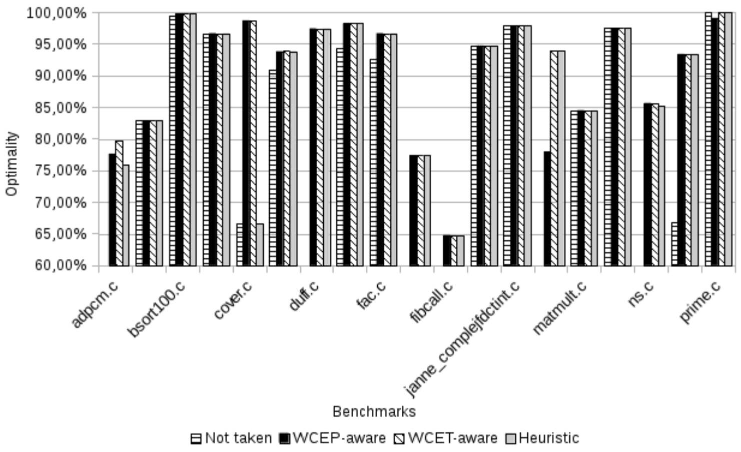


Figure 42 – Graphic comparison of optimalities of Table 15.

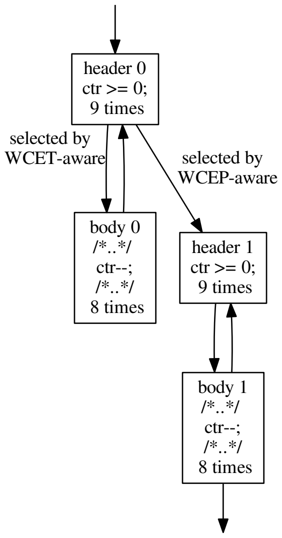


Figure 43 – Control flow graph of Table 7.2 showing the selected prediction of each approach. The text inside the boxes shows the label, operations executed and the number of times that this basic block execute. The compiler uses different registers for *ctr* variable in each loop

8 CONTRIBUTION 3: STATIC GAIN POINT IDENTIFICATION AND GAIN TIME ESTIMATION

8.1 INTRODUCTION

The WCET calculation represents an undecidable problem from the computational point of view. What is done in practice is to estimate upper bounds on its value, using some conservative assumptions. One problem of the WCET is that it is relative to one execution path, called worst-case execution path (WCEP). When a real-time application executes over a path different from the WCEP, its execution time will be certainly smaller than the WCET.

The difference between the WCET of a task and its actual execution time is called gain time (AVILA et al., 2003). Traditionally, gain time identification is performed online by comparing the current execution time with the statically calculated WCET. Early gain time identification is useful to:

- Increase the number of soft real-time tasks that can be accepted at runtime on a system.

- Reduce the power consumption. When a task will not use entirely its processing budget, a power reduction strategy can be employed to lower the processor voltage while still guaranteeing that the task will not overrun its WCET.

In this chapter, we will bring pragmatic answers to the following questions:

- How to statically discover program points where gain time is available without code instrumentation/time measurement? For this intention, we propose a technique that finds specific points of a program (called gain points), where spare times are available before code execution. For these points, we calculate the associated gain times.

- If we want to use code instrumentation in those previously discovered gain points to exploit gain time relative to hardware timing imprecision, what is the WCET to be considered until this point? We must know partial WCET values to use this strategy. Considering this problem, we also propose a technique to estimate these partial WCET values.

As this work is targeted to static analysis, we use the term *gain point* in a slightly different way from literature. For us, a gain point is simply a point where gain time is available and not a point where code instrumentation is injected. All gain points considered in this chapter are automatically discovered and not selected by a programmer. A gain point is always located at the end of a basic block. Among all techniques proposed in this thesis, this is the only that depends solely on WCET analysis, not in compilation.

The remaining of this chapter is organized as follows: Section 8.2 outlines the related work on gain time. We prefer not to include a full chapter covering the related work considering gain time because it represents a small portion of this thesis and is not related to the main topic, which is compiler optimizations. Section 8.3 shows our approach. In Section 8.4 we present a case study involving our technique. Finally, Section 8.5 presents our conclusions and final remarks.

8.2 RELATED WORK

We can separate the works on gain time area in two fields: estimation and utilization. On the first field, (AUDSLEY et al., 1994) introduced the notion of spare time identification using *gain points* and further integration with the *Slack Stealing* algorithm. In this work, the approach is not integrated with an a WCET analyzer.

Considering WCET analyzer, (AVILA et al., 2003) discusses three classes of methods that use static analysis and execution monitoring to explore the spare time of an application. Other works in this field are (HU et al., 2002) (HU et al., 2003), where gain time is reclaimed from real-time java programs. This work integrates WCET analysis for object-oriented programs and gain time reclaiming using annotations placed manually in the code.

Regarding the utilization of gain time, researches on *mixed criticality systems* such as (BATE et al., 2016) depends on the use of gain time to execute tasks with low criticality. At the moment, no work aiming at gain time identification using only static analysis of code was identified in the literature.

8.3 IDENTIFICATION OF GAIN TIME

Our approach operates on the control flow graph of an application and modifies its WCET analysis artifacts, which were explained in detail in Section 5.3. We will explain our technique using the example of Figure 44.

The first step to identify gain time is to search for basic blocks that are candidates to have a gain point. We consider as candidates basic blocks that are out of the WCEP, have a predecessor in WCEP and have a loop bound equal to 1, as we do not explore gain time inside loops. Algorithm 10 summarizes those steps. In this algorithm, $count(bb_i)$ returns the worst-case execution count for a basic block i, lb_i represents the loop bound of i, $n\_pred(bb_i)$ gives the number of predecessor basic blocks of i and $pred(bb_i)$ return the unique predecessor of i.

Algorithm 10 Algorithm used for gain point identification.

```
 1: procedure FIND_GP($cfg$) ▷ Search for a gainpoint on a CFG
 2:     $bb_{gp} \leftarrow nil$
 3:     for all $bb_i \in cfg$ do
 4:         if $count(bb_i) = 0 \wedge lb_i = 1 \wedge n\_pred(bb_i) = 1)$ then
 5:             $bb_j \leftarrow pred(bb_i)$
 6:             if $count(bb_j) \geq 1$ then
 7:                 $bb_{gp} \leftarrow bb_i$
 8:                 break
 9:             end if
10:         end if
11:     end for
12:     return $bb_{gp}$
13: end procedure
```

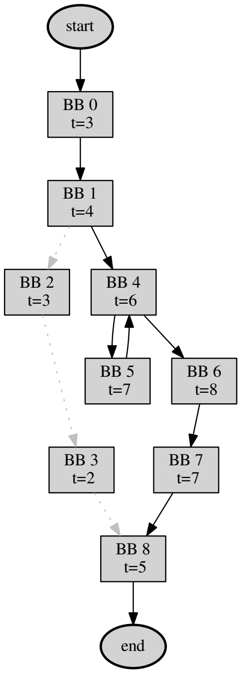


Figure 44 – Control flow graph with a WCEP marked in black.

Figure 45 shows the application of the Algorithm 10 over the example of Figure 44. As we can see, the basic block selected to contain a gain point is the basic block 2 (with a thicker border).

With a basic block selected, we can force the WCET analyzer to include this block on the WCEP, by inserting a constraint to induce the execution count of this basic block to 1. Algorithm 11 shows this approach. In this algorithm, $construct\_ipet\_model(cfg)$ generates an ILP model with the constraints presented in Section 5.3 of Chapter 5, $get\_edge(bb_j, bb_i)$ returns the control flow edge between basic blocks j and i, $insert\_constraint$ adds a new constraint to the IPET model, $solve\_model$ solves the problem using an ILP solver, $update\_counts$ updates the worst-case count of each basic block in the CFG and finally

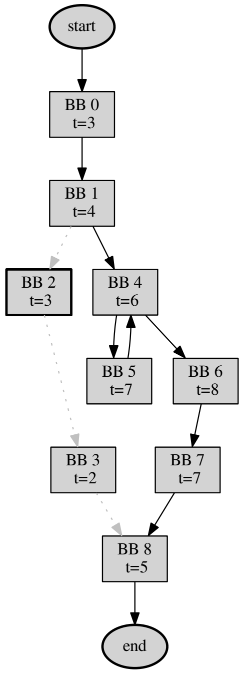


Figure 45 – Control flow graph with a selected basic block (basic block 2, with a thicker border).

objective gives the value of the objective, which is the WCET of the path that includes the gain point.

Algorithm 11 Algorithm used to force a path over a basic block

```
1: procedure FORCE_PATH($cfg$, $bb_i$) ▷ Set predictions to each branch
2:     $ipet\_model \leftarrow construct\_ipet\_model(cfg)$
3:     $bb_j \leftarrow pred(bb_i)$
4:     $dj\_i \leftarrow get\_edge(bb_j, bb_i)$
5:     $insert\_constraint(ipet\_model, dj\_i = 1)$
6:     $solve\_model(ipet\_model)$
7:     $update\_counts(CFG, ipet\_model)$
8:     return $objective(ipet\_model)$
9: end procedure
```

Figure 46 shows the application of the Algorithm 11 over the example of Figure 45. As we can see, the execution path was changed to include the selected basic block.

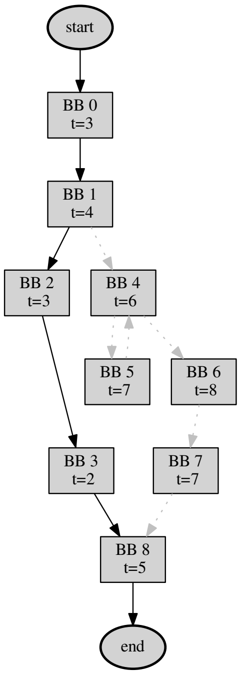


Figure 46 – Control flow graph with the execution path passing over the selected block.

To discover the worst-case execution time until the selected basic block, we must prune the control flow graph to remove all blocks that are: (1) out of the current WCEP and (2) are executed after the selected basic block. For the first category, we must remove those basic blocks with worst-case count equal to 0, i.e., are out of the current WCEP. To the second category, we can simply remove all basic blocks dominated by the selected basic block. Algorithm 12 summarizes the explained approach. In this algorithm, *dom* represents the concept of dominance previously presented and *remove_bb* removes a basic block from a CFG with all associated edges.

Algorithm 12 Algorithm to reduce the control flow graph.

```
1: procedure PATH_UNTIL_GP($cfg$, $bb$) ▷ Reduce the cfg to include basic blocks from entry to bb
2:     for all $bb_i \in cfg$ do
3:         if $count(bb_i) = 0 \vee bb \quad dom \quad bb_i$ then
4:             $remove\_bb(cfg, bb_i)$
5:         end if
6:     end for
7:     $ipet\_model \leftarrow construct\_ipet\_model(cfg)$
8:     $solve\_model(ipet\_model)$
9:     return $objective(ipet\_model)$
10: end procedure
```

Figure 47 shows the application of Algorithm 12 to the example of Figure 46.

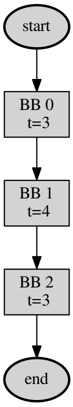


Figure 47 – Reduced control flow graph produced by Algorithm 12.

Finally, Algorithm 13 presents the complete approach to identify a gain point and its estimated gain time. In this algorithm, the variables *initial_wcet*, *through_gp_wcet*, *until_gp_wcet* and *gaintime* represent the initial calculated WCET, the WCET recalculated passing through a gain point, the WCET until the gain point and the estimated gain time for the discovered gain point (bb_{gp}). *do_processor_analysis* is an abstraction of the processor analysis part of a WCET analyzer, where this approach is supposed to be used.

Algorithm 13 Algorithm for gain time estimation.

```
 1: procedure ESTIMATE_GAINTIME($program$)
 2:     $initial\_wcet \leftarrow 0$
 3:     $through\_gp\_wcet \leftarrow 0$
 4:     $until\_gp\_wcet \leftarrow 0$
 5:     $gaintime \leftarrow 0$
 6:     $bb_{gp} \leftarrow nill$
 7:     $cfg \leftarrow reconstruct\_cfg(program)$
 8:     $do\_processor\_analysis(cfg, Program)$
 9:     $ipet\_model \leftarrow construct\_ipet\_model(cfg)$
10:     $solve\_model(ipet\_model)$
11:     $update\_counts(cfg, ipet\_model)$
12:     $initial\_wcet \leftarrow objective(ipet\_model)$
13:     $bb_{gp} \leftarrow$ FIND_GP($cfg$)
14:     if $bb_{gp} \neq nil$ then
15:         $through\_gp\_wcet \leftarrow$ FORCE_PATH($cfg, bb_{gp}$)
16:         $until\_gp\_wcet \leftarrow 0$ PATH_UNTIL_GP($cfg, bb_{gp}$)
17:         $gaintime \leftarrow initial\_wcet - through\_gp\_wcet$
18:     end if
19: end procedure
```

Regarding Algorithm 13, one can note that it discovers only one gain point in its execution. Although, real programs can have more than one gain point with different gain times. To overcome this situation, we can use a data structure to store gain points and iteratively find all available points, ignoring the already discovered ones. For simplicity, we omitted this part of the strategy.

8.4 CASE STUDY: APPLYING ON AN EXAMPLE FROM THE MÄLARDALEN BENCHMARKS

We implemented Algorithms 10 to 13 in our WCET analyzer. As the analyzer constructs only one CFG for the entire program, each function can appear in different parts of the graph, depending on the calling context. As consequence, our technique can discover different gain times for a same program part. As a case study, we present the gain time obtained by applying our strategy to one benchmark from the Mälardalen WCET benchmarks suite (GUSTAFSSON; BETTS, 2010).

8.4.1 Results

We selected the benchmark *qurt.c* because it has a great quantity of available gain points to explore. Under our experimental infrastructure, this benchmark obtained a WCET of 41286 clock cycles in a CFG with 488 basic blocks. The results obtained by our approach are shown in Figure 48. In this figure, we can note that 96 gain points were identified (at the end of 96 basic blocks). For each gain point, this figure shows the WCET until the gain point (*until_gp_wcet* in Algorithm 13) and the associated gain time (*gaintime* in Algorithm 13). We can also notice in this figure that most gain points have a small gain time available, but we still have six gain points with reasonable gain times that can be used. For example, if the execution passes through the bb 35, at the end of it we will have guaranteed a gain time of of 13000 clocks, and in the worst case we will arrive at this point after 15000 clocks counted from the beginning of the execution. Of course, the results will be different if we apply the method to a different benchmark.

8.5 CONCLUSION

Gain time is the difference between the WCET of a task and its actual execution time. A common approach is to identify gain time online by comparing the current measured execution time with the statically calculated WCET. Early gain time identification is useful to increase system utilization at runtime and to save system energy, for example.

We propose in this chapter a strategy to discover gain time using only WCET analysis techniques. Our approach does not rely on code instrumentation. At specific gain points of a program, spare times are available before code execution. To discover spare time before execution, we use integer linear programming technique to force the WCET to include paths passing through gain points.

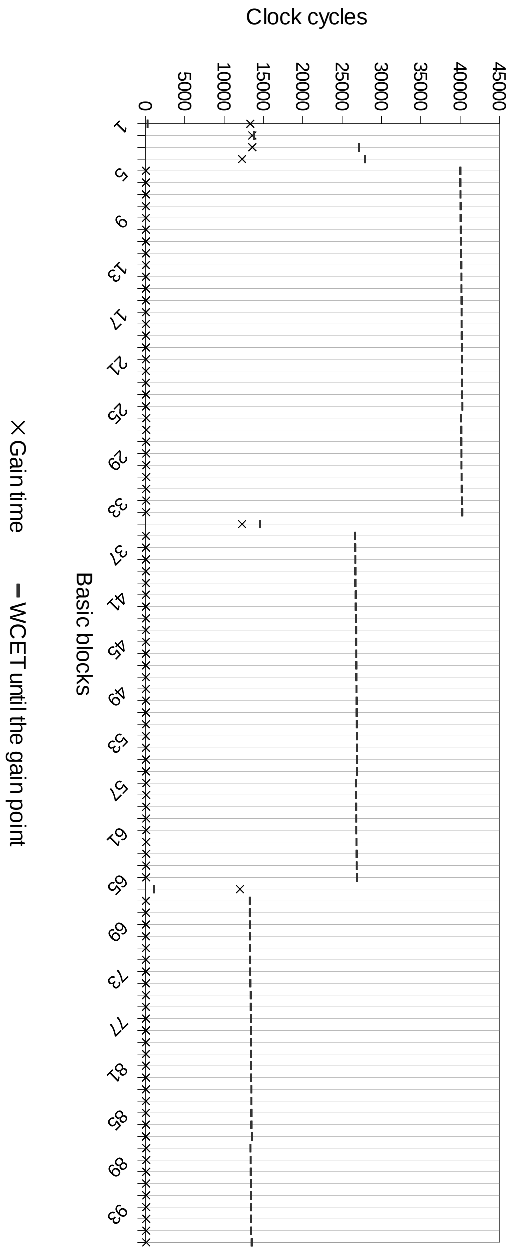


Figure 48 – Gain times and WCET until the gain points

As a case study, we applied our technique on a benchmark of the Mälardalen suite. For the selected example, our technique identified 96 gain points, and calculated the gain time for each of them. In this way, we provided pragmatic answers for questions related to partial WCET calculation until gain points. This information is a good thing if we desire to place code instrumentation on these points. If we want to use gain time without any code instrumentation, our technique gives the amount of time that can be used at an automatically-discovered gain point.

9 FINAL REMARKS

The objective of this thesis was to contribute to aspects related to the compilation for real-time systems, whose primary goal is the WCET reduction or improvement of aspects related to schedulability. This thesis demonstrated that the close coupling of a compiler with a WCET analyzer can benefit both the analysis and synthesis of an executable program to a deterministic architecture. It is also part of our objective to support complex real-time systems in both processor and compiler sides. The complexity of software imposes serious restrictions on the hardware that can be used. This hardware should have sufficient capacity to support the system in question, in addition to be subjected to non-functional project constraints such as cost and energy efficiency.

In order to reach our objective, we followed the academic tendency to using deterministic processors for real-time systems (SCHOEBERL, 2009), (SCHOEBERL et al., 2011), (EDWARDS; LEE, 2007). In this way, we used a deterministic and experimental processor architecture developed using VHDL in another doctoral work (STARKE, 2016) (STARKE et al., 2017). This architecture is based on the ST231 (STMICROELECTRONICS, 2004) processor, which is a member of the ST200 series. Such experimental processor architecture has several hardware mechanisms that give us performance enhancement and determinism together, as parallel execution units, code predication, scratchpad memory and branch prediction.

A new code generator was developed in LLVM as part of the compiler infrastructure. This code generator was implemented because the ISA of the supported processor is not covered by any open source compiler. This code generator implements transformations like packetization, which is used to define which instructions must form each bundle, respecting all dependencies between instructions. We also implemented a code alignment pass because the architecture requires that the bundles are aligned in cache. Another pass was implemented to extract the control flow graph of the application. This graph is useful for WCET analysis. As linker, we used a custom implementation.

Considering WCET analysis, we implemented (in a joint effort with another doctoral work) a WCET analyzer choosing the most appropriate and accurate techniques keeping the basic premise of simplicity of analysis. The chosen techniques basically regards cache analysis, pipeline analysis and finally worst-case path search. The input for the WCET analyzer is a compiled and link-edited program, and a CFG. The CFG is obtained from the compiler, as described above. We used techniques like data flow analysis for cache access classification and implicit path enumeration technique for worst-case path discovery.

Regarding our code generator, it is not possible to perform WCET-oriented optimization using LLVM because it isolates the treatment of each function of a compilation unit. Due to this fact, we cannot optimize the program aiming at WCET reduction using the standard LLVM pass manager because the generated code is only fully materialized at the end of the complete process. To overcome this situation, we implemented two approaches to enable WCET-aware optimizations:

- We modified LLVM to not destroy the data structures for the machine code generated at the end of the pass manager execution. This allows us to make any changes to any function and regenerate the object code file at the end of all LLVM code generation passes. With this LLVM adaptation, any change at the machine code level can be done in any function, followed by an entire object code generation as many times as needed. Using the previously explained LLVM modification with a WCET tool integration, we can perform any WCET-oriented optimization in an iterative way. This approach is inspired in the work of (FALK et al., 2006).

- The second approach was based on the work of (PUSCHNER et al., 2013). In this approach we used a tool to select parts of the program that must be optimized using WCET information as guidance. This tool shares a database with the compiler that is used as communication channel. This database stores facts about the structure of the program and values that specify whether such structure must be touched by an optimization. The tool invokes the compiler to generate the object code and data used as input for the WCET analyzer. After that, WCET information is obtained through the WCET analyzer. Using this information, the planning tool updates the database using optimization heuristics. This process repeats until the stabilization of the WCET.

Considering our objective and the previously explained infrastructure, we made the following contributions:

- The proposition of a different way to perform loop unrolling on data-dependent loops using code predication targeting WCET reduction, because existing techniques only consider loops with fixed execution counts. We also combine our technique with existing unrolling approaches. We introduced an algorithm that performs this code transformation directly at the machine code level (or assembly). We also proposed a strategy that selects which unrolling technique to apply in a per loop basis. For loops with fixed execution counts, we applied the standard technique that unrolls loops using unrolling factors that perfectly divide execution counts to avoid compare and branch instructions. For data dependent loops, we used our predicated or the branch-based approach, depending on the case. We observed in the experiments that the combination of unrolling techniques was able to reduce the WCET of 18 from 33 benchmarks. For six benchmarks we obtained gains above 20%. In the experiments, we also showed that the predicated approach, even with its limited applicability, can exploit cases where the standard approach fails to get WCET reduction.

- Considering static branch prediction schemes, we show that a very small or even no gain can be obtained with new optimization techniques targeted to worst-case execution time reduction. To achieve this conclusion, we evaluated four techniques: the classic approach (BODIN; PUAUT, 2005), a new approach proposed in this thesis, a *not taken* approach and one compiler heuristic. We proposed a new technique because the classic depends on a change on how the WCET is calculated for a determined program. We compared those techniques against the perfect branch predictor. We also showed experimentally that our technique presented better results in terms of WCET reduction. As a third contribution we show that WCET-unaware techniques can also be used in real-time environments because they present good results and low complexity.

- We propose a technique that finds specific points of a program (called gain points, they represent the difference between the WCET of a task and its actual execution time), where there will be an amount of statically estimated gain time in case that path is taken by the execution. The rationale behind this technique is that the WCET is relative to a single execution path, specifically the worst-case execution path (WCEP). When a real-time application executes over a path different from the WCEP, its execution time will be probably smaller than the WCET. As a case study, we present the gain time obtained by applying our strategy to one benchmark from the Mälardalen WCET benchmarks suite. For the selected example, our technique identified 96 gain points, and calculated the gain time for each of them.

9.1 PUBLICATIONS

This thesis generated the following publications in journals:

1. Carminati, A., Starke, R. A., & de Oliveira, R. S. *Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software*. Applied Computing and Informatics. Accepted March 31st, 2017.

   DOI: http://doi.org/10.1016/j.aci.2017.03.002

2. Starke, R. A., Carminati, A., & de Oliveira, R. S. *Evaluation of a low overhead predication system for a deterministic VLIW architecture targeting real-time applications.* Microprocessors and Microsystems. Accepted on November 29th, 2016.

   DOI: http://doi.org/10.1016/j.micpro.2016.11.017.

3. Starke, R. A., Carminati, A., & de Oliveira, R. S. *Evaluating the Design of a VLIW Processor for Real-Time Systems.* ACM Trans actions on Embedded Computing Systems. Accepted on December 7th, 2015. Publication on Vol. 15, No. 3, Article 46.

   DOI: http://dx.doi.org/10.1145/2889490.

Publications in conference proceedings:

1. Starke, R. A., Carminati, A., & de Oliveira, R. S. (2015). *Investigating a four-issue deterministic VLIW architecture for real-time systems.* In 2015 IEEE 13th International Conference on Industrial Informatics (INDIN) (pp. 215–220). Cambrigde - UK, IEEE.

   DOI: http://doi.org/10.1109/INDIN.2015.7281737

Publications in which the doctoral collaborated in the scope of the research group:

1. De Oliveira, R. S., Carminati, & Starke, R. A. *Using an Adversary Simulator to Evaluate Global EDF Scheduling of Sporadic Task Sets on Multiprocessors.* Journal of Parallel and Distributed Computing, Vol. 74, Issue 10, October 2014, pp. 3037–3044.

   DOI: http://doi.org/10.1016/j.jpdc.2014.06.011.

2. De Oliveira, R. S., Carminati, & Starke, R. A. *A Necessary Test for Fixed-Priority Real-Time Multiprocessor Systems based on Lazy-Adversary Simulation* SIMULTECH 2014 – 4th International Conference on Simulated and Modeling Methodologies, Technologies and Applications. Vienna - Austria, 28-30 August, 2014.

3. Carminati, A., de Oliveira, R. S. & Friedrich, L. F. *Exploring the Design Space of Multiprocessor Synchronization Protocols for Real-Time Systems*. Journal of Systems Architecture, ISSN: 1383-7621, 2014.

   DOI: http://doi.org/10.1016/j.sysarc.2013.11.010

9.2 SUGGESTIONS OF FUTURE WORK

This work is not exhaustive in terms of the potential of the experimental infrastructure. We believe that this work can be further explored in the following aspects:

- Scratchpad memory allocation: we do not perform scratchpad memory allocation, we simply use this memory to store the stack of the programs. The application of existent techniques and the proposition of new ones could help to improve WCET reduction.
- Global instruction scheduling: we used the standard LLVM instruction scheduler. due to limited time, we did not explore global instruction scheduling. However, we believe this is an important topic to improve the results presented in this work.
- Refinement on the technique proposed in Chapter : we used a simple heuristic to define unrolling factor. Te use of a more sophisticated heuristic will certainly improve the results.
- Estimation of gain time for more benchmarks: as we only done a proof of concept of our technique, a broad experimental work is a promising next step of the contribution.

During the development of this work, we observed that LLVM (Version 3.3) is a highly optimized compiler, although it has some limitations. The first limitation is the lack of global scheduling support for VLIW architectures. In our original plan, we believed that a VLIW architecture would not be necessary, but we switched to a VLIW one due to requirements of the research group. Another limitation (this limitation is only for us, for LLVM is a positive engineering aspect) is the highly optimized pass manager, which forced us to implement a mechanism to keep data structures of all intermediate code of a program in memory in order to perform WCET-oriented optimizations. This mechanism was implemented directly on the code generator and cannot be directly refactored to the target-independent part of LLVM due to constructive aspects of the compiler. We consider the these two commented topics the biggest challenges we observed on the use of LLVM as a compiler for WCET reduction. Even with those challenges, we were able to integrate our WCET analyzer with LLVM. With our implementation, we observed that WCET reduction through compiler techniques is a viable approach, corroborating similar results that were obtained in the literature. One difference between our infrastructure and other infrastructures used by related works is that we only use custom tools developed in the context of our research group. In this way, we have the smallest possible set of processor instructions, compiler support, boot loader code and WCET analysis support to get a real-time system running with its worst-case execution time known.

BIBLIOGRAPHY

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools*. 2. ed. [S.l.]: Addison Wesley, 2008. 45, 52, 53, 66, 79

ALLEN, J. R. et al. Conversion of control dependence to data dependence. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '83*. New York, New York, USA: ACM Press, 1983. p. 177–189. ISBN 0897910907. 158

ALT, M. et al. Cache behavior prediction by abstract interpretation. In: *Static Analysis*. [S.l.: s.n.], 1996. p. 52–66. 131

AUDSLEY, N. C.; DAVIS, R. I.; BUMS, A. Mechanisms for enhancing the flexibility and utility of hard real-time systems. *Proceedings of the Real-Time Systems Symposium*, p. 12–21, 1994. ISSN 10528725. 14, 208

AVILA, M.; GLAIZOT, M.; PUAUT, I. Impact of automatic gain time identification on tree-based static WCET analysis. In: *Proceeding of the Worst-Case Execution Time Analysis Workshop WCET*. [S.l.: s.n.], 2003. 14, 207, 208

AXER, P. et al. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, v. 13, n. 4, p. 1–37, feb 2014. ISSN 15399087. 33, 39, 70, 181

BATE, I.; BURNS, A.; DAVIS, R. An Enhanced Bailout Protocol for Mixed Criticality Embedded Software. *IEEE Transactions on Software Engineering*, 2016. ISSN 0098-5589. 209

BATE, I.; REUTEMANN, R. Worst-case execution time analysis for dynamic branch predictors. *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS '04).*, 2004. ISSN 1068-3070. 182

BODIN, F.; PUAUT, I. A WCET-Oriented Static Branch Prediction Scheme for Real Time Systems. In: *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. [S.l.]: IEEE, 2005. p. 33–40. ISBN 0-7695-2400-1. 14, 95, 96, 97, 182, 202, 221

BURGUIERE, C.; ROCHANGE, C. On the Complexity of Modeling Dynamic Branch Predictors when Computing Worst-Case Execution Times. In: *Proceedings of the ERCIM/DECOS Workshop On Dependable Embedded Systems*. [S.l.: s.n.], 2007. 182

BURGUIERE, C.; ROCHANGE, C.; SAINRAT, P. A Case for Static Branch Prediction in Real-Time Systems. In: *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*. [S.l.]: IEEE, 2005. p. 33–38. ISBN 0-7695-2346-3. 14, 96, 182, 183, 199, 201

CARMINATI, A.; STARKE, R. A.; OLIVEIRA, R. S. de. Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software. *Journal of Applied Computing and Informatics*, 2017. 155, 176

CHARLESWORTH, A. E. An approach to scientific array processing: the architectural design of the AP-12OB/FPS-164 family. *IEEE Computer*, p. 18–27, 1981. 158

COLIN, A.; PUAUT, I. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems*, v. 18, p. 249–274, 2000. 75, 181, 183, 184, 201

COUSOT, P.; COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77*. New York, New York, USA: ACM Press, 1977. p. 238–252. 73

CYTRON, R. et al. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v. 13, n. 4, p. 451–490, 1991. 46

DASGUPTA, S. The Organization of Microprogram Stores. *ACM Computing Surveys*, v. 11, n. 1, p. 39–65, jan 1979. ISSN 03600300. 58

DEHNERT, J. C.; HSU, P. Y.-T.; BRATT, J. P. Overlapped loop support in the Cydra 5. *ACM SIGARCH Computer Architecture News*, v. 17, n. 2, p. 26–38, 1989. ISSN 01635964. 158

DEVERGE, J.-F.; PUAUT, I. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In: *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS'07)*. [S.l.]: IEEE Computer Society, 2007. p. 179–190. ISBN 0-7695-2914-3. ISSN 1068-3070. 105, 106, 108

EDWARDS, S. A.; LEE, E. A. The case for the precision timed (PRET) machine. In: *Proceedings of the 44th annual conference on Design automation - DAC '07*. New York, New York, USA: ACM Press, 2007. p. 264. ISBN 9781595936271. ISSN 0738100X. 34, 39, 219

ENGBLOM, J. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. Tese (Doutorado) — Faculty of Science and Technology, 2002. 74, 76, 139

ENGBLOM, J.; ERMEDAHL, A. Pipeline timing analysis using a trace-driven simulator. In: *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306)*. [S.l.]: IEEE Computer Society, 1999. p. 88–95. ISBN 0-7695-0306-3. 65, 74

ENGBLOM, J.; ERMEDAHL, A. Modeling complex flows for worst-case execution time analysis. In: *Proceedings of the 21st Real-Time Systems Symposium*. [S.l.]: IEEE Computer Society, 2000. p. 163–174. ISBN 0-7695-0900-2. 153

ERMEDAHL, A. *A modular tool architecture for worst-case execution time analysis*. Tese (Doutorado) — Uppsala University, 2003. 19, 76, 77

FALK, H. WCET-aware register allocation based on graph coloring. In: *Proceedings of the Design Automation Conference, DAC'09. 46th ACM/IEEE*. [S.l.]: IEEE Computer Society, 2009. p. 732–737. ISBN 9781605584973. 98, 108

FALK, H.; KLEINSORGE, J. C. Optimal static WCET-aware scratchpad allocation of program code. In: *Proceedings of the 46th Annual Design Automation Conference on ZZZ - DAC '09*. New York, New York, USA: ACM Press, 2009. p. 732. ISBN 9781605584973. 106, 108

FALK, H.; KOTTHAUS, H. WCET-driven cache-aware code positioning. In: *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems - CASES '11*. New York, New York, USA: ACM Press, 2011. p. 145. ISBN 9781450307130. 40, 101, 104

FALK, H.; LOKUCIEJEWSKI, P. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, v. 46, n. 2, p. 251–300, jul 2010. ISSN 0922-6443. 35, 105, 106

FALK, H.; LOKUCIEJEWSKI, P.; THEILING, H. Design of a WCET-Aware C Compiler. In: *Proceedings of the Workshop on Embedded Systems for Real Time Multimedia*. [S.l.]: IEEE Computer Society, 2006. p. 121–126. ISBN 0-7803-9783-5. 35, 40, 85, 151, 220

FALK, H.; PLAZAR, S.; THEILING, H. Compile-time decided instruction cache locking using worst-case execution paths. In: *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and* *system synthesis - CODES+ISSS '07*. New York, New York, USA: ACM Press, 2007. p. 143. ISBN 9781595938244. 106, 108

FALK, H.; SCHMITZ, N.; SCHMOLL, F. WCET-aware Register Allocation Based on Integer-Linear Programming. In: *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*. [S.l.]: IEEE Computer Society, 2011. p. 13–22. ISBN 978-1-4577-0643-1. 98, 108

FARINES, J.-M.; FRAGA, J.; OLIVEIRA, R. *Sistemas de Tempo Real*. [S.l.: s.n.], 2000. 33

FISHER, J. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30, n. 7, p. 478–490, jul 1981. ISSN 0018-9340. 55, 57

FISHER, J. A.; FREUDENBERGER, S. M. Predicting conditional branch directions from previous runs of a program. In: *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems - ASPLOS-V*. New York, New York, USA: ACM Press, 1992. p. 85–95. ISBN 0897915348. 95, 182

FURBER, S. B. *ARM System Architecture*. [S.l.]: Addison-Wesley Longman Publishing, 1996. ISBN 0201403528. 163

GEVA, R.; MORRIS, D. *IA-64 Architecture Disclosures White Paper*. [S.l.], 1999. 1–20 p. 163

GLOY, N.; SMITH, M. D. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, v. 21, n. 5, p. 977–1027, sep 1999. ISSN 01640925. 102

GUILLON, C. et al. Procedure placement using temporal-ordering information. In: *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems - CASES '04*. New York, New York, USA: ACM Press, 2004. p. 268. ISBN 1581138903. 102

GUSTAFSSON, J.; BETTS, A. The mälardalen WCET benchmarks: Past, present and future. In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'10)*. [S.l.]: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010. p. 136–146. 23, 173, 174, 196, 197, 198, 215

HANXLEDEN, R. v.; KENNEDY, K. Relaxing SIMD control flow constraints using loop transformations. *ACM SIGPLAN Notices*, v. 27, n. 7, p. 188–199, 1992. ISSN 03621340. 158

HEALY, C. et al. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, v. 48, n. 1, p. 53–70, 1999. ISSN 00189340. 74

HEALY, C.; WHALEY, D. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In: *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*. [S.l.]: IEEE Computer Society, 1999. p. 79–88. ISBN 0-7695-0194-X. 76

HU, E. Y.-s.; WELLINGS, A.; BERNAT, G. A Novel Gain Time Reclaiming Framework Integrating WCET Analysis for Object-Oriented Real-Time Systems. In: *In Proceedings of the Second workshop on WCET analysis*. [S.l.: s.n.], 2002. 14, 208

HU, E. Y. S.; WELLINGS, A.; BERNAT, G. Gain time reclaiming in high performance real-time Java systems. In: *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2003*. [S.l.: s.n.], 2003. p. 249–256. ISBN 0769519288. 14, 208

HUANG, Y.; ZHAO, M.; XUE, C. J. WCET-aware re-scheduling register allocation for real-time embedded systems with clustered VLIW architecture. In: *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems - LCTES '12*. New York, New York, USA: ACM Press, 2012. p. 31. ISBN 9781450312127. 35, 99, 108

HWU, W. M. W. et al. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, v. 7, n. 1-2, p. 229–248, may 1993. ISSN 0920-8542. 61, 62

JORDAN, A.; KIM, N.; KRALL, A. IR-level versus machine-level if-conversion for predicated architectures. In: *Proceedings of the 10th Workshop on Optimizations for DSP and Embedded Systems - ODES '13*. New York, New York, USA: ACM Press, 2013. p. 3. ISBN 9781450319058. 158

LATTNER, C.; ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. [S.l.]: IEEE Computer Society, 2004. p. 75–86. ISBN 0-7695-2102-9. 116, 197

LEE, S. et al. Limited preemptible scheduling to embrace cache memory in real-time systems. *Languages, Compilers, and Tools for Embedded Systems, Lecture Notes in Computer Science Volume*, Springer Berlin Heidelberg, Berlin, Heidelberg, v. 1474, p. 51–64, 1998. 131

LENGAUER, T.; TARJAN, R. E. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, v. 1, n. 1, p. 121–141, jan 1979. ISSN 01640925. 129

LI, Y.-T. S.; MALIK, S. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Notices*, v. 30, n. 11, p. 88–98, nov 1995. ISSN 03621340. 76, 129, 139, 140, 144, 153

LI, Y.-T. S.; MALIK, S.; WOLFE, A. Cache modeling for real-time software: beyond direct mapped instruction caches. In: *Proceedings of the 17th IEEE Real-Time Systems Symposium*. [S.l.]: IEEE Computer Society, 1996. p. 254–263. ISBN 0-8186-7689-2. 76, 145

LIANG, Y.; MITRA, T. Improved procedure placement for set associative caches. In: *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems - CASES '10*. New York, New York, USA: ACM Press, 2010. p. 147. ISBN 9781605589039. 102

LIU, C. L.; LAYLAND, J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, v. 20, n. 1, p. 46–61, jan 1973. ISSN 00045411. 35

LIU, I. et al. A PRET microarchitecture implementation with repeatable timing and competitive performance. In: *Proceedings of the 30th International Conference on Computer Design (ICCD)*. [S.l.]: IEEE Computer Society, 2012. p. 87–93. ISBN 978-1-4673-3052-7. 39

LOKUCIEJEWSKI, P.; FALK, H. WCET-driven, code-size critical procedure cloning. In: *Proceedings of the 11th international workshop on Software & compilers for embedded systems*. [S.l.]: ACM Press, 2008. 85, 108

LOKUCIEJEWSKI, P.; FALK, H.; MARWEDEL, P. WCET-driven Cache-based Procedure Positioning Optimizations. In: *Proceedings of the Euromicro Conference on Real-Time Systems*. [S.l.]: IEEE Computer Society, 2008. p. 321–330. ISBN 978-0-7695-3298-1. 102, 103, 108

LOKUCIEJEWSKI, P.; GEDIKLI, F.; MARWEDEL, P. Accelerating WCET-driven Optimizations by the Invariant Path Paradigm. In: *Proceedings of the 12th International Workshop on Software & Compilers for Embedded Systems*. [S.l.]: ACM Press, 2009. 81, 89

LOKUCIEJEWSKI, P.; KELTER, T.; MARWEDEL, P. Superblock-Based Source Code Optimizations for WCET Reduction. In: *Proceedings of the 10th IEEE International Conference on Computer and Information*

Technology. [S.l.]: IEEE Computer Society, 2010. p. 1918–1925. ISBN 978-1-4244-7547-6. 40, 87, 108

LOKUCIEJEWSKI, P.; MARWEDEL, P. Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization. In: *Proceedings of the21st Euromicro Conference on Real-Time Systems*. [S.l.]: IEEE Computer Society, 2009. p. 35–44. ISBN 978-0-7695-3724-5. 40, 108

LOKUCIEJEWSKI, P.; MARWEDEL, P. *Worst-case execution time aware compilation techniques for real-time systems*. [S.l.]: Springer Science & Business Media, 2010. 13, 88, 108, 156, 157, 169, 171, 175, 176, 177

LOKUCIEJEWSKI, P.; MARWEDEL, P. *Worst-Case Execution Time Aware Compilation Techniques for Real- Time Systems*. [S.l.: s.n.], 2011. ISBN 9789048199280. 35, 39, 40, 54, 70, 79, 81, 90, 93, 108, 194

LOUISE, S. Improving branch prediction related WCET abstract interpretation. *Proceedings of the 1st International Workshop on Cyber-Physical Systems, Networks, and Applications (CPSNA'11), Workshop Held During RTCSA 2011*, v. 2, p. 130–133, 2011. ISSN 1533-2306. 182

MAKHORIN, A. GLPK (GNU linear programming kit). 2008. 149

MARREF, A.; BETTS, A. Memory Positioning of Real-Time Code for Smaller Worst-Case Execution Times. In: *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems*. [S.l.]: IEEE Computer Society, 2011. p. 23–32. ISBN 978-1-61284-853-2. 104, 108

MEZZETTI, E.; VARDANEGA, T. A rapid cache-aware procedure positioning optimization to favor incremental development. In: *Proceedings of the IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. [S.l.]: IEEE Computer Society, 2013. p. 107–116. ISBN 978-1-4799-0187-6. 104, 108

MUCHNICK, S. *Advanced compiler design and implementation*. San Francisco: Morgan Kaufmann, 1997. 49

MUELLER, F.; WHALLEY, D. Fast instruction cache analysis via static cache simulation. In: *Proceedings of Simulation Symposium*. IEEE Comput. Soc. Press, 1995. p. 105–114. ISBN 0-8186-7091-6. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=393589>. 131

OTTOSSON, G.; SJODIN, M. Worst-case execution time analysis for modern hardware architectures. In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*. [S.l.]: ACM Press, 1997. 139

PATTERSON, D. A.; LEW, K.; TUCK, R. Towards an efficient, machine-independent language for microprogramming. In: *Proceedings of the 12th annual workshop on Microprogramming*. [S.l.]: ACM Press, 1979. p. 22–35. 58

PATTERSON, J. R. C. Accurate static branch prediction by value range propagation. *ACM SIGPLAN Notices*, v. 30, n. 6, p. 67–78, jun 1995. ISSN 03621340. 95, 182

PLAZAR, S. et al. WCET-driven branch prediction aware code positioning. In: *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems (CASES'11)*. New York, New York, USA: ACM Press, 2011. p. 165. ISBN 9781450307130. 182

POP, S.; YAZDANI, R.; NEILL, Q. Improving GCC's auto-vectorization with if-conversion and loop flattening for AMD's Bulldozer processors. In: *Proceedings of the GCC Developers' Summit 2010*. [S.l.: s.n.], 2010. 158

PRANTL, A.; SCHORDAN, M.; KNOOP, J. TuBound-a conceptually new tool for worst-case execution time analysis. In: *Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis*. [S.l.]: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008. 85

PUAUT, I. WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems. In: *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS'06)*. [S.l.]: IEEE Computer Society, 2006. p. 217–226. ISBN 0-7695-2619-5. 106, 108

PUSCHNER, P. et al. The T-CREST approach of compiler and WCET-analysis integration. In: *Proceedings of the 16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*. [S.l.: s.n.], 2013. p. 1–8. ISBN 978-1-4799-2111-9. 151, 152, 220

SCHNEIDER, J.; FERDINAND, C. Pipeline behavior prediction for superscalar processors by abstract interpretation. In: *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems - LCTES '99*. New York, New York, USA: ACM Press, 1999. p. 35–44. ISBN 1581131364. 73

SCHOEBERL, M. Time-Predictable Computer Architecture. *EURASIP Journal on Embedded Systems*, v. 2009, p. 1–17, 2009. ISSN 1687-3955. 34, 219

SCHOEBERL, M. et al. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In: LUCAS, P. et al. (Ed.). *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011. (OpenAccess Series in Informatics (OASIcs), v. 18), p. 11–21. ISBN 978-3-939897-28-6. ISSN 2190-6807. 34, 219

SCO. *System V Application Binary Interface - DRAFT*. [S.l.], 2013. 128

SMITH, J. E. A study of branch prediction strategies. In: *Proceedings of the 8th annual symposium on Computer Architecture*. [S.l.: s.n.], 1981. p. 135–148. ISBN 1581130589. 182

SREEDHAR, V. C.; GAO, G. R. A linear time algorithm for placing ϕ-nodes. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, New York, USA: ACM Press, 1995. p. 62–73. ISBN 0897916921. 123

STAPPERT, F.; ALTENBERND, P. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, v. 46, n. 4, p. 339–355, jan 2000. ISSN 13837621. 76

STAPPERT, F.; ERMEDAHL, A.; ENGBLOM, J. Efficient longest executable path search for programs with complex flows and pipeline effects. In: *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems - CASES '01*. New York, New York, USA: ACM Press, 2001. p. 132. ISBN 1581133995. 76

STARKE, R. A. *Design and evaluation of a VLIW processor for real-time systems*. Tese (Doutorado) — Federal University of Santa Catarina, 2016. 23, 41, 109, 219, 241, 246

STARKE, R. A.; CARMINATI, A.; OLIVEIRA, R. S. D. Evaluating the Design of a VLIW Processor for Real-Time Systems. *ACM Transactions on Embedded Computing Systems*, v. 15, n. 3, p. 1–26, 2016. ISSN 15399087. 111

STARKE, R. A.; CARMINATI, A.; OLIVEIRA, R. S. D. Evaluation of a low overhead predication system for a deterministic VLIW architecture targeting real-time applications. *Microprocessors and Microsystems*, v. 49, p. 1–8, 2017. 109, 219

STMICROELECTRONICS. *ST231 Core and Instruction Set Architecture Manual*. [S.l.]: MCDT Documentation Group, 2004. 109, 219

SUHENDRA, V.; MITRA, T.; ROYCHOUDHURY, A. WCET Centric Data Allocation to Scratchpad Memory. In: *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*. [S.l.]: IEEE Computer Society, 2005. p. 223–232. ISBN 0-7695-2490-7. 105, 106, 108

THEILING, H. ILP-based interprocedural path analysis. *Lecture Notes in Computer Science*, v. 2491, p. 349–363, 2002. 76

THESING, S. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models Dissertation*. Tese (Doutorado) — Universität des Saarlandes, 2004. 73

THIELE, L.; WILHELM, R. Design for Timing Predictability. *Real-Time Systems*, Kluwer Academic Publishers, Norwell, MA, USA, v. 28, n. 2-3, p. 157–177, nov 2004. ISSN 0922-6443. 37

UNGERER, T. et al. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, v. 30, n. 5, p. 66–75, sep 2010. ISSN 0272-1732. 39

WAN, Q.; WU, H.; XUE, J. WCET-aware data selection and allocation for scratchpad memory. In: *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems - LCTES '12*. New York, New York, USA: ACM Press, 2012. p. 41. ISBN 9781450312127. 106, 108

WILHELM, R. et al. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 28, n. 7, p. 966–978, jul 2009. ISSN 0278-0070. 70

WILHELM, R. et al. The worst-case execution-time problem–overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, v. 7, n. 3, p. 1–53, apr 2008. ISSN 15399087. 19, 34, 35, 67, 68, 69, 75, 182, 186

WILLIAM, L.; BARBARA, G. Pointer-induced aliasing: a problem taxonomy. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. [S.l.]: ACM Press, 1991. 66, 126

ZHAO, W. et al. Improving WCET by applying worst-case path optimizations. *Real-Time Systems*, v. 34, n. 2, p. 129–152, jun 2006. ISSN 0922-6443. 13, 40, 79, 91, 92, 108, 156, 157, 176

ZHAO, W. et al. Improving WCET by applying a WC code-positioning optimization. *ACM Transactions on Architecture and Code Optimization*, v. 2, n. 4, p. 335–365, dec 2005. ISSN 15443566. 40, 100, 108

Annex

ANNEX A – INSTRUCTIONS OF THE TARGET ARCHITECTURE

Operand	Encoding Bits	Type
bcond	25..23	Predicate ($*br*)
bdest	20..18	Predicate ($*br*)
btarg	22..0	Immediate
dest	17..12	Register ($*r*)
idest	11..6	Register ($*r*)
isrc2	20..12	Immediate
scond	23..21	Predicate ($*br*)
src1	5..0	Register ($*r*)
src2	11..6	Register ($*r*)
pc		Program counter

Table 16 – List of operation operands (STARKE, 2016)

Mnemonic	Semantic
ADD	dest $=$ *signext32* (src1) $+$ *signext32* (src2)
ADD_I	idest $=$ *signext32* (src1) $+$ *signext32* (isrc2)
SUB	dest $=$ *signext32* (src1) - *signext32* (src2)
SUB_I	idest $=$ *signext32* (src1) - *signext32* (isrc2)
SHL	dest $=$ *signext32* (src1) $\ll$ src2 (4..0)
SHL_I	idest $=$ *signext32* (src1) $\ll$ *signext32* (isrc2 (4..0))
SHR	dest $=$ *signext32* (src1) $\gg$ src2 (4..0)
SHR_I	idest $=$ *signext32* (src1) $\gg$ *signext32* (isrc2 (4..0))
SHRU	dest $=$ *signext32* (src1) $\gg$ src2 (4..0)
SHRU_I	idest $=$ *signext32* (src1) $\gg$ *signext32* (isrc2 (4..0))
AND	dest $=$ *signext32* (src1) & *signext32* (src2)

AND_I	idest = *signext32* (src1) & *signext32* (isrc2)
ANDC	dest = *signext32* (src1) & *signext32* (src2)
ANDC_I	idest = *signext32* (src1) & *signext32* (isrc2)
OR	dest = *signext32* (src1) | *signext32* (src2)
OR_I	idest = *signext32* (src1) | *signext32* (isrc2)
ORC	dest = *signext32* (src1) | *signext32* (src2)
ORC_I	idest = *signext32* (src1) | *signext32* (isrc2)
XOR	dest = *signext32* (src1) ^*signext32* (src2)
XOR_I	idest = *signext32* (src1) ^*signext32* (isrc2)
MAX	dest = *signext32* (src1) > *signext32* (src2) ? src1 : src2
MAX_I	idest = *signext32* (src1) > *signext32* (isrc2) ? src1 : isrc2
MAXU	dest = *zeroext32* (src1) > *zeroext32* (src2) ? src1 : src2
MAXU_I	idest = *zeroext32* (src1) > *zeroext32* (isrc2) ? src1 : isrc2
MIN	dest = *signext32* (src1) <*signext32* (src2) ? src1 : src2
MIN_I	idest = *signext32* (src1) <*signext32* (isrc2) ? src1 : isrc2
MINU	dest = *zeroext32* (src1) <*zeroext32* (src2) ? src1 : src2
MINU_I	idest = *zeroext32* (src1) <*zeroext32* (isrc2) ? src1 : isrc2
SXTB	idest = *signext32* (src1(7..0))
SXTH	idest = *signext32* (src1(15..0))
ZXTB	idest = *zeroext32* (src1(7..0))
ZXTH	idest = *zeroext32* (src1(15..0))
ADDCG	dest = *signext32* (src1) + *signext32* (src2) + *zeroext1*(scond) ; bdest = carry bit
SUBCG	dest = *signext32* (src1) - *signext32* (src2) + *zeroext1*(scond) – 1 ; bdest = carry bit

CMPEQ_R	dest(1) = *signext32* (src1) == *signext32* (src2)
CMPEQ_B	bdest = *signext32* (src1) == *signext32* (isrc2)
CMPEQ_IR	dest(1) = *signext32* (src1) == *signext32* (isrc2)
CMPEQ_IB	bdest = *signext32* (src1) == *signext32* (isrc2)
CMPNE_R	dest(1) = *signext32* (src1) == *signext32* (src2)
CMPNE_B	bdest = *signext32* (src1) != *signext32* (isrc2)
CMPNE_IR	dest(1) = *signext32* (src1) != *signext32* (isrc2)
CMPNE_IB	bdest = *signext32* (src1) != *signext32* (isrc2)
CMPGE_R	dest(1) = *signext32* (src1) >= *signext32* (src2)
CMPGE_B	bdest = *signext32* (src1) >= *signext32* (isrc2)
CMPGE_IR	dest(1) = *signext32* (src1) >= *signext32* (isrc2)
CMPGE_IB	bdest = *signext32* (src1) >= *signext32* (isrc2)
CMPGEU_R	dest(1) = *zeroext32* (src1) >= *zeroext32* (src2)
CMPGEU_B	bdest = *zeroext32* (src1) >= *zeroext32* (isrc2)
CMPGEU_IR	dest(1) = *zeroext32* (src1) >= *zeroext32* (isrc2)
CMPGEU_IB	bdest = *zeroext32* (src1) >= *zeroext32* (isrc2)
CMPGT_R	dest(1) = *signext32* (src1) > *signext32* (src2)
CMPGT_B	bdest = *signext32* (src1) > *signext32* (isrc2)
CMPGT_IR	dest(1) = *signext32* (src1) > *signext32* (isrc2)
CMPGT_IB	bdest = *signext32* (src1) > *signext32* (isrc2)
CMPGTU_R	dest(1) = *zeroext32* (src1) > *zeroext32* (src2)
CMPGTU_B	bdest = *zeroext32* (src1) > *zeroext32* (isrc2)
CMPGTU_IR	dest(1) = *zeroext32* (src1) > *zeroext32* (isrc2)
CMPGTU_IB	bdest = *zeroext32* (src1) > *zeroext32* (isrc2)
CMPLE_R	dest(1) = *signext32* (src1) <= *signext32* (src2)

CMPLE_B	bdest = *signext32* (src1) <= *signext32* (isrc2)
CMPLE_IR	dest(1) = *signext32* (src1) <= *signext32* (isrc2)
CMPLE_IB	bdest = *signext32* (src1) <= *signext32* (isrc2)
CMPLEU_R	dest(1) = *zeroext32* (src1) <= *zeroext32* (src2)
CMPLEU_B	bdest = *zeroext32* (src1) <= *zeroext32* (isrc2)
CMPLEU_IR	dest(1) = *zeroext32* (src1) <= *zeroext32* (isrc2)
CMPLEU_IB	bdest = *zeroext32* (src1) <= *zeroext32* (isrc2)
CMPLT_R	dest(1) = *signext32* (src1) <*signext32* (src2)
CMPLT_B	bdest = *signext32* (src1) <*signext32* (isrc2)
CMPLT_IR	dest(1) = *signext32* (src1) <*signext32* (isrc2)
CMPLT_IB	bdest = *signext32* (src1) <*signext32* (isrc2)
CMPLTU_R	dest(1) = *zeroext32* (src1) <*zeroext32* (src2)
CMPLTU_B	bdest = *zeroext32* (src1) <*zeroext32* (isrc2)
CMPLTU_IR	dest(1) = *zeroext32* (src1) <*zeroext32* (isrc2)
CMPLTU_IB	bdest = *zeroext32* (src1) <*zeroext32* (isrc2)
ANDL_R	dest(1) = *zeroext32* (src1) && *zeroext32* (src2)
ANDL_B	bdest = *zeroext32* (src1) && *zeroext32* (isrc2)
ANDL_IR	dest(1) = *zeroext32* (src1) && *zeroext32* (isrc2)
ANDL_IB	bdest = *zeroext32* (src1) && *zeroext32* (isrc2)
NANDL_R	dest(1) = (*zeroext32* (src1) && *zeroext32* (src2))
NANDL_B	bdest = (*zeroext32* (src1) && *zeroext32* (isrc2))
NANDL_IR	dest(1) = (*zeroext32* (src1) && *zeroext32* (isrc2))
NANDL_IB	bdest = (*zeroext32* (src1) && *zeroext32* (isrc2))
NORL_R	dest(1) = (*zeroext32* (src1) || *zeroext32* (src2))

NORL_B	bdest = (*zeroext32* (src1) || *zeroext32* (isrc2))
NORL_IR	dest(1) = (*zeroext32* (src1) || *zeroext32* (isrc2))
NORL_IB	bdest = (*zeroext32* (src1) || *zeroext32* (isrc2))
ORL_R	dest(1) = *zeroext32* (src1) || *zeroext32* (src2)
ORL_B	bdest = *zeroext32* (src1) || *zeroext32* (isrc2)
ORL_IR	dest(1) = *zeroext32* (src1) || *zeroext32* (isrc2)
ORL_IB	bdest = *zeroext32* (src1) || *zeroext32* (isrc2)
SLCT_R	dest = (scond == 1 ? src1 : src2)
SLCT_I	dest = (scond == 1 ? src1 : isrc2)
SLCTF_R	dest = (scond == 0 ? src1 : src2)
SLCTF_I	dest = (scond == 0 ? src1 : isrc2)
BR	pc = (scond == 1 ? *signext32* (pc) + *signext32* (btarg))
BRF	pc = (scond == 0 ? *signext32* (pc) + *signext32* (btarg))
MULL	Dest = (32 lower bits) *signext32* (src1) * *signext32* (src2)
MULL64H	Dest = (32 higher bits) *signext32* (src1) * *signext32* (src2)
MULL64HU	Dest = (32 higher bits) *signext32* (src1) * *signext32* (src2)
DIV_R	dest = *signext32* (src1) % *signext32* (src2)
DIV_Q	dest = *signext32* (src1) / *signext32* (src2)
DIV_RU	dest = *zeroext32* (src1) % *zeroext32* (src2)
DIV_QU	dest = *zeroext32* (src1) / *zeroext32* (src2)
CALL	pc = src1 R63 = pc
ICALL	pc = btarg R63 = pc
GOTO	pc == *signext32* (pc) + *signext32* (btarg)
IGOTO	pc = *signext32* (pc) + *signext32* (btarg)
IMML	imm for previous operation is: *signext32*(btarg $\ll$ 9 + isrc2)

IMMR	imm for next operation is: *signext32*(btarg $\ll$ 9 + isrc2)
LDW	idest = *signext32* (mem [*signext32* (src1) + *signext32* (isrc2))
LDH	idest = *signext32*(mem [*signext32* (src1) + *signext32* (isrc2) (15..0))
LDHU	idest = mem [*signext32* (src1) + *signext32* (isrc2) (15..0)
LDB	idest = *signext32*(mem [*signext32* (src1) + *signext32* (isrc2) (7..0))
LDBU	idest = mem [*signext32* (src1) + *signext32* (isrc2) (7..0)
STW	mem [*signext32* (src1) + *signext32* (isrc2)) = src2
STH	mem [*signext32* (src1) + *signext32* (isrc2)) = src2 (15..0)
STB	mem [*signext32* (src1) + *signext32* (isrc2)) = src2 (7..d0)
HALT	halt the cpu
PAR_ON	enable predication complete mode
PAR_OFF	disable predication complete mode
PRELD	direct next branch to (*signext32* (pc) + *signext32* (btarg))

Table 17 – List of supported operations (STARKE, 2016)