FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# On the Impact of Classical Program Optimization on the WCET-Behaviour

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Computational Intelligence**

eingereicht von

**Benjamin Eizinger**
Matrikelnummer 0726743

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Dr. Jens Knoop
Mitwirkung: Dipl. Ing. Stefan Hepp

Wien, 10.08.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

# On the Impact of Classical Program Optimization on the WCET-Behaviour

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Computational Intelligence**

by

**Benjamin Eizinger**
Registration Number 0726743

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Dr. Jens Knoop
Assistance: Dipl. Ing. Stefan Hepp

Vienna, 10.08.2015

(Signature of Author) (Signature of Advisor)

# Erklärung zur Verfassung der Arbeit

Benjamin Eizinger
Tokiostrasse 3/1/16, 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

| | |
|---|---|
| (Ort, Datum) | (Unterschrift Verfasser) |

# Abstract

In real time systems every program has a time bound in which the execution has to be finished and therefore the worst case execution time (WCET) of such a program is a safety critical, non functional requirement as it obviously has to be below the time bound.

To be sure of not violating the time bound usually a processor more powerful than necessary is employed. This is obviously a waste of resources and as such it is desirable to reduce the WCET.

There is a vast array of available compiler program optimizations in modern compilers and it would be beneficial to harness their optimization potential. Those optimizations are usually developed with improvement of the average case execution time in mind and their effect on the WCET has not been studied yet in detail.

This work aims to shed some light on the effects of classical compiler optimizations on the WCET of a selection of benchmark programs. To examine the effect of optimizations on the WCET of programs 33 different benchmarks have been selected from the well established Mälardalen Benchmark Suite and the WCET Tool Challenge 2014 Benchmark Suite.

The WCET of the unoptimized programs have been compared to the WCET of the optimized cases. A safe optimization is one that guarantees to not degrade the WCET.

The main results are: firstly, code motion in general is not safe, secondly, by replacing simple subexpressions (i.e. incrementing an index variable by one and constant expressions) the WCET usually cannot be improved and thirdly, function inlining appears to be a safe optimization as it never resulted in a degradation of the WCET.

# Kurzfassung

Echtzeitsysteme beschränken die Ausführungszeit, die ein Programm maximal in Anspruch nehmen darf. Die worst case execution time (WCET), also die maximale Ausführungszeit eines Programms, muss unterhalb dieser Grenze liegen.

Es ist wünschenswert, die WCET eines Programms so gering wie möglich zu halten, da so Ressourcen gespart werden können. Dies lässt sich z.B. durch automatische Programmoptimierung bewerkstelligen. Moderne Compiler besitzen eine Vielzahl an Optimierungen, deren Auswirkungen auf die WCET noch vergleichsweise unerforscht sind.

In der vorliegenden Arbeit werden nun die Auswirkungen klassischer Optimierungen auf die WCET untersucht. Dazu wird eine Reihe etablierter Benchmarks verwendet - insgesamt 33 Programme aus den Mälardalen Benchmarks und der WCET Tool Challenge 2014.

Analysiert wurden 36 Optimierungen, indem die WCET jedes Programms zuerst unoptimiert kalkuliert wurde, anschliessend noch einmal nach jeder Optimierung. Das Verhältnis von 'optimierter' zu unoptimierter WCET gibt Aufschluss auf den Effekt der Optimierung.

Die wichtigsten Ergebnisse der durchgeführten Arbeit lassen sich wie folgt zusammenfassen:

Das Verschieben von Instruktionen, also das Vor- oder Nachreihen der Instruktion im Programmfluss, ist keine sichere Operation, da die WCET dadurch verschlechtert werden kann.

Die Ersetzung einfacher Ausdrücke, z.B. das Hochzählen einer Indexvariablen, durch eine temporäre Variable bringt keine Verbesserung der WCET.

Durch Funktions-Inlining wurde die WCET nie verschlechtert, weshalb sie als sichere Optimierung angeführt wird.

# Contents

**Kurzfassung** **v**

**1 Introduction** **1**
1.1 Motivation . . . . . 1
1.2 Problem Statement . . . . . 1
1.3 Aim of the Work . . . . . 2
1.4 Contributions of the Work . . . . . 2
1.5 Methodological approach . . . . . 2
1.6 Differentiation to Related Work . . . . . 2
1.7 Structure of the Work . . . . . 3

**2 State of the Art** **5**
2.1 Estimation of the WCET or Timing Analysis . . . . . 5
2.2 Reducing the WCET . . . . . 8

**3 Optimizations** **11**
3.1 Terminology . . . . . 11
3.2 Standard Optimizations . . . . . 11
3.3 Assignment Motion . . . . . 16
3.4 Partial Dead Code Elimination . . . . . 19
3.5 Combination of AM and PDCE . . . . . 20

**4 Framework** **23**
4.1 Assignment Motion and Partial Dead Code Elimination Implementation . . . . 23
4.2 Compilation . . . . . 26

**5 Evaluation** **27**
5.1 Benchmarks and Timing . . . . . 27
5.2 The Effect of Optimizations . . . . . 28
5.3 Discussion of Results . . . . . 39

**6 Conclusion** **43**
6.1 Summary . . . . . 43
6.2 Risks . . . . . 43

6.3 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

**A Benchmarks** **45**

**B Listings** **47**

**List of Tables** **47**

**List of Figures** **47**

**List of Listings** **48**

**Bibliography** **49**

CHAPTER 1

# Introduction

## 1.1 Motivation

Embedded systems often have to meet timing constraints, they have to finish their execution within the given time budget [12, 22]. This subset is called real-time systems. They are of great importance in automobiles, flight control systems, aircraft avionics, robotics and many more [51]. As such the worst-case execution time (WCET) is a safety critical, non functional requirement of such real-time systems. As only with the knowledge of how long it takes the program in the worst case to respond, it can be guaranteed that the response meets the deadline [22].

Regardless of the approach towards obtaining a WCET estimate, it has to be overestimated to guarantee a safe bound. As a consequence most hardware resources are oversized [12]. So by providing better (tighter) WCET estimates or reducing the WCET of a program, there is potential to reduce the oversized hardware and provide a cheaper and more efficient product.

But obtaining an estimate WCET is a tricky task and the results of the 2006 WCET Tool challenge show an overestimation of the WCET of up to 289.33% [52]. On the other hand, applying compiler optimizations to programs is risky as there is no guarantee that there will not be a performance degradation [41, 58].

## 1.2 Problem Statement

As we have seen reduction of the WCET is a relevant task to real-time systems. This can be done by optimizing the program, either by hand (which would be error prone and a lot of work) or by automatic compiler optimizations. There is a huge number of compiler optimizations available [2, 42]. Many of them are available for the most common compilers, but their effect on the WCET has not been studied thoroughly so far.

## 1.3 Aim of the Work

This works aims to shed light on the question what effect different optimizations have on the WCET. Are there optimizations that are universally good or bad? This information in turn can enable practitioners in selecting which optimizations they want to use, thus helping with the reduction of WCET. Furthermore this information could be used to help with optimization sequence selection, see: [30, 33, 41].

## 1.4 Contributions of the Work

I was able to show that, even though one would naively assume that certain optimizations will always have a positive effect on the WCET, this is not necessarily the case as the factors that influence the WCET are not trivial. By simply changing the order instructions are executed in, the WCET could be affected. Furthermore a wide range of optimizations' effect on the WCET has been investigated.

## 1.5 Methodological approach

Various compiler optimizations available in the LLVM compiler for the Patmos framework[1] [48], as well as three additional optimizations (Partial Dead Code Elimination [28], Assignment Motion [29] and their combination Assigment Placement [27]) that have been implemented, for the purpose of this thesis, in the Rose Compiler Framework[2], are analysed with the AbsInt aiT WCET estimation tool[3] on selected benchmarks from the Mälardalen WCET Benchmark suite[4] and WCET Tool Challenge 2014 Benchmark suite[5]. See appendix A for a comprehensive list of benchmark programs. Benchmarks were excluded because they were too large to be analysed effectively, contained pointer arithmetic or unstructured code.

The WCET of the optimized programs is then compared to the WCET of their unoptimized counterparts.

From this data we then are able to draw conclusions about the effect of traditional compiler optimizations on the WCET.

## 1.6 Differentiation to Related Work

Most of the related work was done on a custom compiler infrastructure and/or only for a very few selected optimizations. This work aims at providing a broad overview of optimizations that are available in the very popular LLVM compiler, as well as implementing further optimizations that have not been investigated so far. The compiler is already used in real time systems (for

[1]https://github.com/t-crest
[2]http://rosecompiler.org/
[3]http://www.absint.com/ait/
[4]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[5]http://www.irit.fr/wiki/doku.php?id=wtc:benchmarks:start

example in the Patmos project) and therefore the work has a broader audience. Furthermore by using the aiT WCET estimator we use an industrial strength WCET analysis tool.

## 1.7 Structure of the Work

This work is structured as follows: Chapter 2 gives an overview of the state of the art in WCET estimation and optimization. Chapter 3 will introduce the optimizations that are being evaluated. Chapter 4 explains the framework for the custom optimizations (Partial Dead Code Elimination, Assignment Motion and Assignment Placement) as well as the evaluation. Moreover, WCET timing pipelines are explained and the used benchmarks are introduced. Chapter 5.1 gives the results of the evaluations and discusses reasons for the witnessed effects. Finally Chapter 5.3 gives a summary of the results obtained and discusses them in the context of existing research results.

CHAPTER 2

# State of the Art

## 2.1 Estimation of the WCET or Timing Analysis

### Overview

This chapter presents some of the state-of-the-art developments in the fields of WCET estimation and optimization. First an outline will be given of how the WCET can be estimated, what problems WCET estimation faces and different methods for optimizing code to produce a lower WCET are presented.

A timing analysis tool must be able to compute the time spent by the hardware on the given instructions (the path). To compute the WCET it furthermore has to consider all possible paths that can be taken in the program to compute a safe estimate. From this we can define two main subtasks for WCET estimation, the computation of valid paths (Control-Flow Analysis) and the computation of execution times of paths (Processor-Behaviour Analysis).

### Control-Flow Analysis

WCET analysis tools usually work on the binary level [57], as only the actual executable holds all the relevant information and the actual WCET is dependent on the specific processor architecture. This poses a challenge for the Control Flow Graph (CFG) reconstruction by having dynamic calls via virtual function calls or dynamically computed jumps (e.g. in switch statements) in the code. But most assembly languages usually label their switch case statements so this is normally not a problem [57].

The computed CFG is then used to determine the feasible paths. Not every path in the CFG is possible as subsequent conditions or input data can make some paths infeasible. Calculating the set of feasible paths as precisely as possible is crucially important, as the analysis becomes more tight [16] because we do not consider paths that cannot be taken by the program. One part of the path calculation is the determination of loop and recursion bounds (if possible). Most programs spend their time looping or in a recursion [57], therefore calculating exact bounds excludes infeasible paths and increases the tightness of the result [16].

Another possibility is for the user to manually annotate loop bounds and other flow information, but this approach is labour intensive and error prone [18]. One method to improve manual annotations is by verifying them via model checking [44]. Changes to the code can invalidate those annotations or the user can simply make an error while annotating, etc. A fully automatic approach would therefore be preferable.

### Processor-Behaviour Analysis

The aim of Processor Behaviour Analysis is to obtain information about the components that influence execution times, e.g. caches, pipelines, etc. This is important as execution times of single instructions and therefore of basic blocks and paths are very much dependent on the context they are executed in [57]. What this means is we cannot assume that the WCET of the subsequently executed blocks A and B

$$w_{AB}$$

is

$$w_{AB} = w_A + w_B.$$

This is due to the superscalar nature of modern microprocessors that increase their performance (e.g. caches, pipelines, out of order execution [56], etc.). Exactly those features are what make the analysis hard [5, 21], and this leads to the phenomenon that local worst cases do not necessarily lead to the global worst case.

### Challenges

The phenomenon mentioned in the previous section is called ‘timing anomaly’ [5]. Informally this means that some local worst case (e.g. a cache miss) can actually be the better case globally.

This happens due to the fact that timing analysis usually abstracts the underlying hardware/software system in order to make the analysis feasible [46, 57]. Additionally one does not usually know all the input data and therefore some state information is missing during the analysis. This missing information leads to non-determinism if decisions depend on the missing information [57]. The ramification of this is that the results are different depending on assumptions about that missing state information.

For example, if we do not know if the next instruction is in the cache or not, we either start with a cache load or cache miss. Naively we would assume that the cache miss will yield the worse execution time, but this may not be the case [45].

The problem that arises from this is that we simply cannot assume the worst case for every statement, but instead have to track all the information about cache state, branch predictions etc. throughout the program to obtain a safe WCET estimate. This state space can grow very quickly [56] and hence this becomes hard to do efficiently. It has been proven that a necessary criterion for timing anomalies is the existence of different resource allocations [55]. I.e. a sequence of instructions can produce multiple different resource allocations (when it needs which functional unit of a processor), simply by changing the execution time of one instruction.

#### Timing Calculation

Finally the Control-Flow information and Processor information are combined to obtain a WCET estimate. How this is calculated depends on the actual methods used in the Control-Flow Analysis and Processor-Behaviour Analysis phases.

### Static Methods

Static Methods do not rely on execution or simulation of the actual hardware, but use code analysis techniques, like static program analysis (see. [8, 43] for details), to obtain properties about the program state at various points in the program. Those are combined with a model of the system to obtain WCET estimates.

#### Solution to Control-Flow Analysis

The main goal of Control-Flow Analysis is to compute possible execution paths. This usually has to be done on binary executable level as opposed to source code level where it would be easier [57]. The two main points, loop bound calculation and infeasible path calculation can be done in numerous ways.

Li and Malik [35] developed a method called Implicit Path Enumeration Technique (IPET) that generates linear constraints about the program flow to solve the WCET with integer programming.

Healey et al. [19, 20] employ a data flow algorithm to calculate loop bounds.

Thesing [53] combines interval-based abstract interpretation with pattern matching to calculate loop bounds from the machine code.

Gustafsson et al. [16] developed their algorithm 'Abstract Execution', a form of symbolic execution based on abstract interpretation, to obtain loop bound and infeasible path constraints. The WCET is then calculated by IPET. Later [10] they expanded upon their algorithm to calculate the WCET by including a time variable to be incremented in each basic block. This removes the possible exponential running time that is a problem when using IPET.

Knoop et al. [26] are using pattern matching, satisfiability modulo theory and rewriting of loops into simpler forms to automatically obtain loop bounds.

#### Solution to the Processor-Behaviour Analysis

As the execution time for most instructions depends on the state of the processor [57], it is important to obtain information about the state the processor is in at each instruction in the program.

This is usually accomplished by a Data Flow Analysis that is combined with an abstract model of the processor to yield invariants that are true at certain instructions. It may be the case that multiple invariants can be computed, each valid for a specific set of execution paths, also called the calling context [57].

The calculated invariants now represent the known information about the state of the processor's functional units like cache, pipeline, etc. which is vital in calculating the correct and tight WCET for instructions.

### Measurement Based Methods

Measurement Based Methods usually perform the timing analysis by measuring (parts of) the source codes execution time. This measurement can be done either on the real hardware or on a cycle accurate simulator of the hardware. This has the advantage over static approaches that no complex timing model of the target processor has to be developed, and the resulting timing tool is much easier to employ on a new processor. Another advantage of measurement-based approaches is that usually there is a certain trade-off between complexity (and time required) of the analysis task and precision by varying the number of measurement cases [25].

Obviously running end-to-end measurements on all possible input states is infeasible, as the number of paths is exponential in the size of the code [25]. Because of that usually only a subset of the possible input states is run on sub-graphs of the whole program.

Those measurements of sub-graphs are then combined with the result of the Control-Flow Analysis, which can be done the same way as in static analysis, to obtain an execution time estimate. This can only produce estimates, and not safe bounds as can static analysis, because we might never encounter the worst case for the measurements [57]. We could of course increase the number of measurements but that will only increase the certainty of our estimate but will never guarantee that we have actually found the worst case. Another possibility is to assume worst case initial states for all inputs, but this is not an easy task for processors that exhibit timing anomalies since there is no obvious initial worst state that triggers the worst-case time behaviour. But for processors with simple timing models it is possible to obtain bounds that way [57].

Kirner et al. [25, 56] use measurement of program segments and static analysis for test data generation and Control-Flow analysis to obtain WCET estimates by implicit path enumeration. They define a coverage criterion that describes the number of required measurements to be taken.

Schaefer et al. [47] use a similar approach. They measure execution times for basic blocks, and combine those with static analysis of the CFG and a simple processor model to estimate the WCET. They validate their measurements by estimating the WCET of basic blocks with static analysis and if the measured execution time is significantly lower than the predicted one, they generate new test data (until either the predicted WCET is achieved or user intervention is required).

Hansen et al. [17] employ Extreme Value Theory [4, 15] for WCET estimation, furthermore they can give a probability for exceeding their predicted WCET. Their algorithm works with a set of execution time traces to compute a WCET estimate that stays below the given exceedance probability.

## 2.2 Reducing the WCET

Reducing the WCET by applying compiler optimizations automatically is very desirable for real time systems [58]. A lower WCET means for example that less powerful processors can be used (that can still meet the now lowered timing requirement), thereby reducing costs and power consumption. While most modern compilers have a lot of different optimizations available, they usually were developed with the goal of improving the average case. This leads to the problem that traditional compiler optimizations are not safe to use on software systems where strict timing requirements have to be met, as they could make the WCET even worse.

This section will introduce some of the methods that have been used to deal with this problem.

One approach is to search for sequences of optimizations that will together reduce the WCET, to verify the safety of the transformation timing analysis is usually employed to verify that the WCET has not been impaired.

Another approach is WCET aware optimization, which uses a tight integration of a timing model/information into the compiler so the optimizer can employ this timing information to facilitate smarter optimizations.

Lastly traditional compiler optimization have been used to reduce the WCET [34, 38, 50].

### WCET Aware Optimization

Optimizations that can exploit timing information have been developed as a strategy to handle the WCET optimization problem specifically.

Zhao et al. [59] proposed an algorithm for improving the WCET path by various code transformations (superblock formation, path duplication, code sinking, loop unrolling, etc.). They use a timing analyser to calculate the worst case path information, as well as to evaluate if a transformation that would increase code size, actually improves the WCET. They concluded that this approach holds some merit but at the same time poses the problem that there is another path that can quickly become the worst case path and therefore the gain is limited.

Falk et al. [12, 13] developed a worst case aware C compiler. They integrated the timing analyser aiT[1] into their C compiler to be able to exploit timing information for optimizations. By now they have integrated a range of WCET aware optimizations into the compiler, like procedure cloning [37], code positioning [36], loop unrolling [39], function inlining [40], register allocation [11, 14], etc. By applying the compiler on an example they can show that the optimizations can improve upon the standard optimization level O2 by up to 70% in terms of WCET reduction.

### Finding Optimization Sequences

It has been recognized that not every sequence of compiler optimizations works most effectively for every program [30]. This is doubly true when the WCET of a program has to be optimized.

The search for the best sequence of optimizations has seen attention of researchers [30, 31, 33]. This can involve genetic algorithms [30], sample sizing [33] or machine learning [1]. Those methods have been shown to find sequences of optimizations that are up to 10% better (in terms of execution time) than if optimized by a standard optimization sequence (e.g. O3) [30].

Zhao et al. [60] developed a genetic algorithm for finding optimization sequences. The optimization sequence is encoded into a string that is used as the basis for the genetic algorithm to work on. Then point mutations and crossover are employed to generate new candidate solutions. They managed to improve upon the standard sequence by an average of 7% and up to 21%.

[1]`http://www.absint.com/`

Lokuciejewski et al. [41] developed this concept for WCET optimization further. They employed evolutionary multi-object algorithms (EMO) to find trade-offs between at most two different objectives (in their case average case execution time, worst case execution time and code size). The optimization sequence is encoded into a string that is used as the basis for the EMO algorithm to work on. EMOs work similar to evolutionary algorithms in that they employ crossover and point mutation to explore the possible solution space. They managed to achieve an improvement over O3 of about 30%, but as a result the resulting code size is increased by 133%. One of the overall results is that they have to choose between improving the WCET or code size, but cannot improve on both.

### Influence of Traditional Compiler Optimizations

Schwarzer [50] investigated the effect of traditional compiler optimizations on the WCET of programs. He could show that most optimizations do not have an effect on the WCET on their own, a few of made the WCET worse and only one showed significant improvements of the benchmark cases across the board.

Lokuciejewski et al. [38] describe the optimizations Dead Code Elimination (DCE) and Common Subexpression Elimination (CSE) for superblocks. Additionally they also investigate the effect on the WCET of standard CSE/DCE and conclude that both can lead to increases in the WCET after the optimisation, and that their superblock transformation is worthwhile as it always decreases the WCET of their benchmarks.

Li et al. [34] also investigate the effect of traditional compiler optimizations on the WCET of benchmark programs. They are able to show that some optimizations have a large positive effect on the WCET and that some optimizations can lead to a degradation of the WCET performance. But they find that most optimizations do not have an influence on the WCET in their evaluation setting.

CHAPTER 3

# Optimizations

This section will introduce the optimizations that are being evaluated.

## 3.1 Terminology

- Safe Optimization
  An optimization that is guaranteed to not impair the WCET.
- CFG
  control flow graph, a graph representation of the programs execution paths.
- AST
  abstract syntax tree, an abstract syntactic representation of the program code.
- dead variable assignment
  An assignment to a variable is dead, if the variable is not used after the assignment statement, or if the variable is assigned a new value before it is used again.

## 3.2 Standard Optimizations

In this section we will review the standard compiler optimizations that are being evaluated, I will give references to literature where possible. The compilation is done with the LLVM compiler[1] introduced in this thesis [32]. For a full list of available optimizations as well as further information we refer the reader to the LLVM documentation.[2]

- Aggressive Dead Code Elimination (*adce*)

[1] `http://llvm.org/`
[2] `http://llvm.org/docs/Passes.html`

Tries to eliminate dead code, in contrast to Dead Code Elimination (dce) it will assume that a value is dead unless proven otherwise.

- Promote 'by reference' to scalars (*argpromotion*)

  This transforms reference arguments that are only loaded to value arguments. I.e. instead of using a reference it will then use the value of the argument directly.

- Profile Guided Basic Block Placement (*block-placement*)

  This transformation tries to put blocks that are executed frequently at the begin of the function and therefore hopefully increases the number of fall-through conditional branches.

- Optimize for code generation (*codegenprepare*)

  Tries to optimize functions for code generation.

- Merge Duplicate Global Constants (*constmerge*)

  Merges duplicate global constants together into one constant.

- Simple constant propagation (*constprop*)

  Constant propagation looks for constant expressions and replaces those with a single constant. So e.g.:

  ```
  a = 4;
  b = a + 5;
  ```

  is transformed into:

  ```
  a = 4;
  b = 9;
  ```

  As the value of $a$ is constant at the computation site of $a + 5$, we can replace it by just the constant value 9.

- Dead Code Elimination (*dce*)

  Similar to Dead Instruction Elimination but rechecks instructions that were used by eliminated instructions to see if they are newly dead. See [23] for details.

- Dead Argument Elimination (*deadargelim*)

  Deletes dead arguments (i.e. ones that are not used in the function) as well as those arguments that are passed into a function and used only as dead arguments.

- Dead Instruction Elimination (*die*)

  Removes all instructions that are obviously dead (i.e. can never be reached).

- Dead Store Elimination (*dse*)

  Removes basic block local dead variables.

- Early common subexpression elimination (*early-cse*)

  Reduces identical (sub)expressions that can be combined to reduce computation times. E.g:

  ```
  a = b * c + g;
  d = b * c * e;
  ```

  The subexpression $b*c$ is calculated twice, now if we store the value in a separate variable:

  ```
  tmp = b * c;
  a = tmp + g;
  d = tmp * e;
  ```

  We can save time if retrieving tmp is faster than the calculation of $b*c$. See [7] for details.

- Function Integration/Inlining (*inline*)

  Performs inlining bottom up of functions into callees.

- Combine redundant instructions (*instcombine*)

  Combines instructions to create fewer, simpler instructions. Also algebraic simplification is performed. E.g:

  ```
  y = x + 1;
  z = y + 1;
  ```

  Would be combined into:

  ```
  z = x + 2;
  ```

  Additionally it will perform more canonizations. For a complete list see the LLVM documentation.

- Interprocedural constant propagation (*ipconstprop*)

  Like constant propagation, just interprocedurally.

- Interprocedural Sparse Conditional Constant Propagation (*ipsccp*)

  An interprocedural version of sccp.

- Loop Invariant Code Motion (*licm*)

  Tries to move as much loop invariant code out of a loop. Either by sinking it after the loop body, or in front of the loop header, whichever can be done safely. Further information [3].

- Recognize loop idioms (*loop-idiom*)

  Tries to transform simple loops into non loop structures.

- Simplify instructions in loops (*loop-instsimplify*)

  Tries to simplify instructions in loop bodies.

- Loop Strength Reduction (*loop-reduce*)

  Reduces the strength on array references inside loops that use the loop induction variable as one or more of their components.

- Canonicalize natural loops (*loop-simplify*)

  Simplifies natural loops to have a simpler form and enable subsequent optimizations. After the transformation all loops will satisfy the following:

  - There is a single non critical edge into the loop header.
  - There is only one back edge in the loop.
  - All exit blocks from the loop only have predecessors in the loop.

- Unroll loops (loop-unroll)

  Simple loop unrolling. See [3] for details.

- Promote memory to register (*mem2reg*)

  Promotes memory references to be register references and constructs a "pruned" SSA form.

- MemCpy Optimization (*memcpyopt*)

  Transformation that aim at eliminating memcpy calls or transform sets of stores into memsets.

- Merge Functions (*mergefunc*)

  Looks for equivalent functions and folds them.

- Unify function exit nodes (*mergereturn*)

  Ensures that functions have only one return statement.

- Partial Inliner (*partial-inliner*)

  Performs inlining of certain constructs like the inlining of an if statement that fully surrounds the body of a function.

- Reassociate expressions (*reassociate*)

  Reorders the operands of commutative expressions into an order that improves constant propagation.
  E.g:

  ```
  4 + (x + 5) => x + (4 + 5)
  ```

- Sparse Conditional Constant Propagation (*sccp*)

  Constant propagation and merging. See [54] for details.

- Simplify the CFG (*simplifycfg*)

  Simplifies the CFG (control flow graph) by deleting basic blocks without predecessors and merging basic blocks that only have one predecessor which has only one successor, and more transformations.

- Code sinking (*sink*)

  Moves instructions into successor blocks if admissible. This is done so that in paths where they are not required they do not get executed.

- Tail Call Elimination (*tailcallelim*)

  Standard tail call elimination with a few additions. Tail call elimination transforms self recursive function calls that are immediately followed by an return instruction into a loop.

## 3.3 Assignment Motion

Introduced in [29], the optimization aims to eliminate (partially) redundant computation of expressions/assignments. I will now give a short overview of how Assignment Motion works and what can be achieved with it.

Knoop et al. describe three steps [29]:

1. Initialization: Introducing temporaries
2. Assignment Motion: Eliminating partially redundant assignments
3. Final Flush: Eliminating unnecessary initializations of temporaries

In the initialization phase every assignment of the form: $x := t$; is replaced by the sequence: $h_t := t; x := h_t;$, where $h_t$ is a unique fresh temporary variable that is associated with the term $t$ and inserted wherever the the term $t$ is used. Let's demonstrate this on the following simple example:

```
1: if a + b < c + d:
2:     x = a + b;
3: else:
4:     y = c - x;
5: y = c - x;
```

As we can see the assignment at line 5 $y = c - x$; is partially redundant, as it will already have been executed when the program progresses along the else branch of the if statement.

Applying the initialization step to this program would yield:

```
 1: h1 = a + b;
 2: h2 = c + d;
 3: if h1 < h2:
 4:     h1 = a + b;
 5:     x = h1;
 6: else:
 7:     h3 = c - x;
 8:     y = h3;
 9: h3 = c - x;
10: y = h3;
```

The next phase "Assignment Motion" consists of an arbitrary sequence of the following two steps:

- Redundant Assignment Elimination
- Assignment Hoisting

The Redundant Assignment Elimination is based on a bit-vector data flow analysis that calculates for every assignment pattern $\alpha$ (of the form $v = t$;) if it is redundant at instruction $i$, and if such an assignment pattern $\alpha$ is redundant immediately before an instruction of $\alpha$ that instruction will be eliminated.

An assignment pattern $\alpha \equiv x = t$ is redundant at an instruction $i$ if every computation path from the start of the program $s$, goes through an instruction $k$ containing $\alpha$, and neither $x$ nor any variables in $t$ are modified between those two occurrences of $\alpha$.

If we look again at the example program we can see that the assignment $h1 = a + b$; at line 4 is redundant as: neither h1, a nor b are modified between the lines 1 and 4. If we remove that assignment we get the following program:

```
1: h1 = a + b;
2: h2 = c + d;
3: if h1 < h2:
4:     x = h1;
5: else:
6:     h3 = c - x;
7:     y = h3;
8: h3 = c - x;
9: y = h3;
```

Assignment Hoisting now moves assignments to earlier program points. Of course such a move has to be semantics preserving and therefore it has to be admissible. A hoisting of assignment $x = t$; can be blocked by the instruction $i$ if a variable in $t$ is modified by the instruction $i$ or $x$ is modified or used in $i$. Furthermore hoisting will never introduce an assignment $x = t$; into loops and can only move out of branches (loops/ifs) if every branch contains an unblocked occurrence of the assignment $x = t$;.

In our example we can hoist the assignments $h3 = c - d$; and $y = h3$; at lines 8 and 9 respectively, into the if statement (and not further as same assignment to $x$ blocks further movement). This will result in the following program:

```
 1: h1 = a + b;
 2: h2 = c + d;
 3: if h1 < h2:
 4:     x = h1;
 5:     h3 = c - x;
 6:     y = h3;
 7: else:
 8:     h3 = c - x;
 9:     h3 = c - x;
10:     y = h3;
11:     y = h3;
```

Now we remove the redundant assignment in lines 9 and 11 and obtain the following program:

```
1: h1 = a + b;
2: h2 = c + d;
3: if h1 < h2:
4:     x = h1;
5:     h3 = c - x;
```

```
    y = h3;
else:
    h3 = c - x;
    y = h3;
```

As further code hoisting no longer produces any changes (i.e. enables further elimination potential) we proceed to the last phase in the algorithm. The Final Flush now moves all assignments to temporary variables to the 'latest' safe execution point and then eliminates them if their left hand side (i.e. the temporary variable name) is only used once immediately after the assignment. This means our example will look like this after the Final Flush:

```
h1 = a + b;
if h1 < c + d:
    x = h1;
    h3 = c - x;
    y = h3;
else:
    h3 = c - x;
    y = h3;
```

We could eliminate $h2$ as it was only used once in the if condition, but we cannot eliminate $h1$ as it is not only used in the if condition, but in the assignment of $x$ as well. Compared to the original program we have a bit more code but the following inefficiencies were eliminated:

- The double execution of $y = c - x$; when taking the else path is gone.
- $a + b$ is calculated only once and used in the if condition as well as in the assignment of $x$

For further details we refer to: [29].

## 3.4 Partial Dead Code Elimination

Introduced in [28], the optimization aims at eliminating all (partially) dead variable assignments. A dead variable assignment is an assignment that will never be used in a program, either the variable itself will never be used, or another assignment to the variable is made before the variable is used again. I will now give a short overview of how Partial Dead Code Elimination works and what can be achieved with it.

The Partial Dead Code Elimination algorithm has two parts:

- Assignment Sinking
- Dead Code Elimination

The sinking of variables is analogous to the hoisting of variables described in 3.3, just with moving a statement to their latest possible execution point. In order for a sinking movement to be admissible it has to be semantics preserving. So any instruction $i$ will block any sinking of an assignment $x = t$; that modifies any variable in $t$ and modifies or uses the variable $x$. Let's consider the following program:

```
1: y = a + b;
2: if x < c + d:
3:     y = x + 1;
4: else:
5:     y = y + 1;
```

As we can see, the assignment $y = a + b$; is partially dead with respect to the then branch of the if statement as the value in $y$ will be overwritten with $x + 1$ without ever being used. So we perform sinking and get the following program:

```
1: if x < c + d:
2:     y = a + b;
3:     y = x + 1;
4: else:
5:     y = a + b;
6:     y = y + 1;
```

Now we can apply the next step, the elimination of dead variable assignments. An occurrence of an assignment pattern $\alpha \equiv x = t$; in the instruction $i$ is dead, if the variable $x$ is dead, i.e., on every path from $i$ to the end of the program the usage of $x$ on the right-hand side is preceded by a modification of $x$.

In our example this clearly applies to the assignment $y = a+b$; on line 2. For the assignment on line 5 it does not hold, as the variable $y$ is itself used in the assignment on line 6. If we now apply the elimination we get the following result:

```
1: if x < c + d:
2:     y = x + 1;
3: else:
4:     y = a + b;
5:     y = y + 1;
```

For further details see [28].

## 3.5 Combination of AM and PDCE

In the two previous sections we have seen the optimization potentials of Assignment Motion and Partial Dead Code Elimination. Another possibility now is to execute both optimizations in sequence. Which can improve the performance further. This is demonstrated below.

Let us consider the end result of the Assignment Motion example from page 19:

```
h1 = a + b;
if h1 < c + d:
    x = h1;
    h3 = c - x;
    y = h3;
else:
    h3 = c - x;
    y = h3;
```

We can see that the assignments $h3 = c - x$; and $y = h3$; are present in both branches of the if statement. When we now apply Partial Dead Code Elimination on the program we would sink both assignments and get the following program (after removing the dead/redundant assignments and the Final Flush phase from Assignment Motion):

```
h1 = a + b;
if h1 < c + d:
    x = h1;
y = c - x;
```

This not only reduces the size of code, but also removes the temporary variable $h3$, so we can see that it clearly can improve the code even further. This approach also called Assignment Placement [27] just has one drawback. While Partial Dead Code Elimination and Assignment Motion are optimal (i.e. they always reach a 'global optimum' that cannot be improved any further, this property is lost if both are interleaved [27]. That means the resulting program is dependent on the order in which Assignment Motion and Partial Dead Code Elimination are performed, though it is still locally optimal.

CHAPTER 4

# Framework

To date most evaluations of optimizations for WCET have been done on specialized compilers that have not a very wide practical distribution. In this work we use the LLVM compiler and are interested in how good the compiler optimizations work in the WCET setting. Furthermore two additional optimizations of interest, Assignment Motion and Partial Dead Code Elimination (explained in sections 3.3 and 3.4 respectively) are not implemented in LLVM so they had to be implemented first. An outline of the implementation as well as the pipeline for optimizing and compiling the benchmarks (section 4.2) is given in this chapter.

## 4.1 Assignment Motion and Partial Dead Code Elimination Implementation

The two optimizations ‘Partial Dead Code Elimination’ [28] and ‘Assignment Motion’ [29], are implemented with the ROSE[1] and SATIrE [49] (now a part of the ROSE compiler) frameworks.

ROSE and SATIrE are used to read in a C source file, generate an AST and CFG, transform the AST and then create a C source from the AST again. A custom simplified CFG and data-flow analysis [24] framework were created and used to implement both optimizations, as they employ data-flow analysis exclusively.

The process is as follows, see Figure 4.1 for a visual representation:

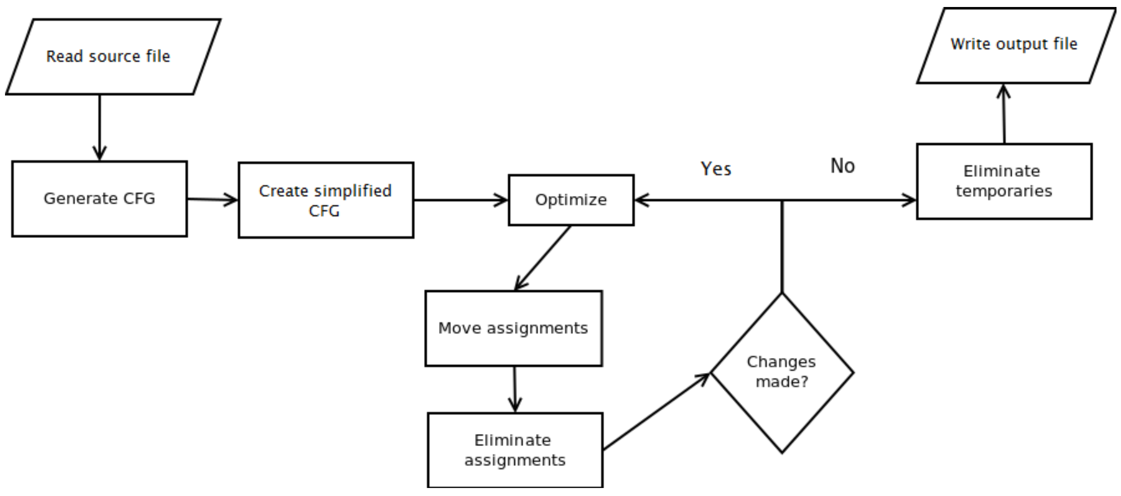
1. Read source file (by ROSE)
2. Generate CFG (by SATIrE)
3. Create simplified CFG from SATIrE CFG
4. Perform canonization

[1]www.rosecompiler.org

5. Perform optimizations
6. Elimination of temporaries
7. Output AST as program again (by ROSE)

Points four, five and six warrant some more explanation.

**Figure 4.1:** Process of the AM and PDCE implementations



## Canonization

This step is performed to simplify the handling of the AST/CFG as well as to unlock further optimization potential:

- Shortening of expressions. If an expression has more than one operation its subexpressions are split and stored into temporary variables. Also called 3-adress form in [28] and required for the Assignment Motion algorithm. For example:

  ```
  x = (y + z) - a;
  ```

  Would be split into the following:

  ```
  temp1 = y + z;
  x = temp1 - a;
  ```

  This can enable additional optimization potential, e.g: when the expression y + z is used more than once as a subexpression. Additionally when writing the data-flow analysis we can be sure that the right hand side of every expression is composed of only one binary operator.

- Transformation of for loops into while loops.
- Replacing ‘shorthand’ expressions with their long form. All assignment expressions of the form:

  ```
  x OP= expr;
  ```

  are replaced by:

  ```
  x = x OP expr;
  ```

  Where OP is an operation in {+, -, *, \}.

  Furthermore:

  ```
  x++; => x = x + 1;
  x--; => x = x - 1;
  ```

  This is all done to simplify the CFG.
- Splitting of declarations. If more than one declaration is done in one statement, it is split up into two statements (again for simplifying the resulting CFG):

  ```
  int x, y;
  ```

  is split into:

  ```
  int x;
  int y;
  ```
- Separation of declaration and initialization. If the declaration and initialization are done in the same statement they are split:

  ```
  int x = 0;
  ```

  is separated into:

  ```
  int x;
  x = 0;
  ```

  This is done to simplify the CFG and to enable optimizations (as the assignment x=0 may be dead).

### Optimization step

Both optimizations work the same way:

1. Perform data-flow analysis to determine moveable assignments (hoisting or sinking)
2. Apply the move operations
3. Perform data-flow analysis to determine removable assignments
4. Remove statements that were deemed redundant/dead from CFG
5. Repeat

This loop is performed until no more changes (with exception of code sinking/hoisting) are seen.

The combined optimization of AM and PDCE works slightly different. It starts with performing AM until it terminates, then calls PDCE until that terminates and then starts with AM again until none of them can produce further nontrivial changes.

### Elimination of Temporaries

The canonization introduces temporaries (when generating the 3-address form). Which are removed where possible. That means the following:

1. We sink every assignment to a temporary as far down the execution path as possible.
2. If the assignment is immediately followed by the single use of the temporary then we can remove the assignment and declaration, then replace it's usage by the right hand side of the temporary variables assignment.

## 4.2 Compilation

For obtaining the binary executables the llvm compiler of the patmos toolchain is used, a simple script calls the compiler with an optimization as argument on a benchmark program. Afterwards with the help of the Platin tool[2] from the patmos toolchain, the aiT tool is called and the WCET result retrieved and stored in a CSV file. This is performed for every (standard) optimization and benchmark.

For the custom implementation of 'Assignment Motion', 'Partial Dead Code Elimination' and 'Assignment Placement', a slightly different approach has to be taken as they cannot be used via compiler arguments. The source benchmark programs are first run through the optimizer which outputs the transformed C code. This code then can be run through the analysis pipeline pretty much the same way as described before, we just have to disable all compiler optimizations with the -O0 flag.

[2]https://github.com/t-crest/patmos-llvm/tree/master/tools/platin

CHAPTER 5

# Evaluation

## 5.1 Benchmarks and Timing

### WCET Timing

WCET timing is performed with the aiT tool from AbsInt[1] on the Patmos [48] processor. aiT employs abstract interpretation of the binary executable and is not dependent on the source code and reports deterministic WCET cycle counts. Therefore compiler optimizations can reorganize the code without impact upon the analyseability of the benchmark. It performed well in the WCET Toolchain Challenge of 2006 [52].

### Used Benchmarks

The benchmark programs that were evaluated are a subset of the Mälardalen benchmarks[2] and some benchmarks of the 2014 WCET Tool Challenge[3]. Benchmarks were excluded because they were too large to analyse effectively, contained pointer arithmetic or unstructured code. One custom program was added that represents one particular function that is used in multiple Mälardalen benchmarks. See Appendix A for a full list and short description of each benchmark program.

The Mälardalen benchmarks were chosen because they are used by the WCET community as benchmarks for evaluating WCET computation accuracy.

The original programs were modified slightly to obtain uniform and analysable programs. The list of changes is as follows:

- Console outputs were removed
- All main methods return exit code 0

[1]http://www.absint.com/ait/

[2]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[3]http://www.irit.fr/wiki/doku.php?id=wtc:benchmarks:start

- Critical result variables were made volatile (so that the actual calculation of the result was not optimized away as that would defeat the purpose)

## 5.2 The Effect of Optimizations

Most optimizations have no effect on the WCET by themselves. This is probably largely due to the fact that there just is no optimization potential for that particular kind of optimization. For example *tailcallelim* shows no effect on the WCET, this is due to the fact that none of the benchmark programs has a recursive tail call that could be eliminated. Therefore this chapter will focus only on those optimizations that show an effect on at least some of the benchmark programs.

Now I describe groups of optimizations that show similar effects on the WCET of the benchmark programs. The values provided are relative WCET numbers, i.e. the WCET of the optimized version divided by the WCET of the unoptimized version.

### Statistics

Table 5.1 contains an aggregated form of the WCET performance measurements. The average effect is average of the absolute effect (both positive and negative) on the benchmarks where the optimization does have an effect. The column ‘WCET degradations’ is the percentage of benchmarks (of those that were affected), that showed a degradation in their WCET after the optimization. The column ‘WCET improvement’ is the same just for an improvement of the WCET.

### Loop Transformations

The loop transformations (see 3.2), Loop Invariant Code Motion (licm), Recognize Loop Idioms (loop-idiom), Simplify instructions in loops (loop-instsimplify), Loop Strength Reduction (loop-reduce), Canonicalize natural loops (loop-simplify) and Unroll loops (loop-unroll) do not have a strong effect on the WCET, see Table 5.2.

As we can see from the table, every optimization has the same effect on the benchmark ‘compress.c’, this benchmark contains goto statements, and is the only one to do so. Investigating the benchmarks suggest that the effect witnessed is because all optimizations contain the same basic transformations. The increase in the WCET could be explained by the disruption of the cache, pipeline or other such influence on the processor state during execution.

Another observation to note is the increase in WCET for the case of ‘licm’ and ‘ndes.c’. Even though one would naively assume that the removal of loop invariant code from the loop body would reduce the WCET, it actually increases it in this case. The resulting increase in WCET is again probably due to cache disruption effects or something similar.

Overall the conclusion is that loop transformations are not safe on their own. On their own they reorganize the code maybe a bit, but appear not to be able to reduce the WCET by themselves.

| Optimization | affected benchmarks | average effect | WCET degradations | WCET improvements |
|---|---|---|---|---|
| AP | 45.45% | 3.58% | 40.00% | 60.00% |
| DCE | 45.45% | 4.16% | 40.00% | 60.00% |
| AM | 45.45% | 3.65% | 46.67% | 53.33% |
| block-placement | 66.67% | 0.74% | 68.18% | 31.82% |
| break-crit-edges | 63.64% | 5.71% | 28.57% | 71.43% |
| codegenprepare | 54.55% | 2.35% | 16.67% | 83.33% |
| early-cse | 66.67% | 2.99% | 45.45% | 54.55% |
| inline | 57.58% | 8.42% | 0.00% | 100.00% |
| instcombine | 96.97% | 2.92% | 40.63% | 59.38% |
| licm | 12.12% | 1.38% | 100.00% | 0.00% |
| loop-idiom | 9.09% | 0.60% | 100.00% | 0.00% |
| loop-instsimplify | 9.09% | 0.60% | 100.00% | 0.00% |
| loop-reduce | 9.09% | 0.61% | 100.00% | 0.00% |
| loop-simplify | 9.09% | 0.60% | 100.00% | 0.00% |
| loop-unroll | 9.09% | 0.60% | 100.00% | 0.00% |
| mem2reg | 93.94% | 12.62% | 51.61% | 48.39% |
| mergefunc | 24.24% | 1.66% | 100.00% | 0.00% |
| mergereturn | 21.21% | 1.23% | 100.00% | 0.00% |
| reassociate | 3.03% | 7.55% | 100.00% | 0.00% |
| simplifycfg | 57.58% | 3.96% | 21.05% | 78.95% |
| sink | 24.24% | 2.49% | 75.00% | 25.00% |

**Table 5.1:** Statistics for optimizations

### Code Reorganization

Next, we consider those optimizations that reorganize or modify the control flow of a program. In particular Code Sinking (sink), Merge Functions (mergefunc), Unify Function Exit Nodes (mergereturn) and Reassociate Expressions (reassociate) for further details see Section 3.2.

As we can see in Table 5.3 most effects are increases in the WCET of the benchmark. This is expected for the given optimizations as most of them have to insert statements to perform their optimizations (mergefunc, mergereturn).

Reassociate expressions and code sinking are interesting cases as one would expect no change from them as they simply reorder the already present instructions. But this does have an effect, as we can see in the case of code sinking, both a positive and negative effect on the WCET. The most obvious reason is that this effect can be explained by cache disruption/enabling.

| | Optimizations | | | | | |
|---|---|---|---|---|---|---|
| Benchmarks | licm | loop-idiom | loop-instsimplify | loop-reduce | loop-simplify | loop-unroll |
| bs.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| bsort100.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| cnt.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| compress.c | 1.016 | 1.016 | 1.016 | 1.016 | 1.016 | 1.016 |
| coop.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| cover.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| crc.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| duff.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| edn.c | 1.011 | 1.001 | 1.001 | 1.002 | 1.001 | 1.001 |
| fac.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| fdct.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| fft1.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| fibcall.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| insertsort.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| janne_complex.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| jfdctint.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| lcdnum.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| lms.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| loop3.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| ludcmp.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| matmult.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| matmult_32_32.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| matmult_128_128.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| minmax.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| minver.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| ndes.c | 1.028 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| ns.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| nsichneu.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| qsort-exam.c | 1.001 | 1.001 | 1.001 | 1.001 | 1.001 | 1.001 |
| qurt.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| select.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| sin_func.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| sqrt.c | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |

**Table 5.2:** Relative WCET values for Loop Optimizations

Overall, as these first optimizations are not intended to be used solely, but rather as optimization potential enabling transformations, it is not surprising that they do not have a positive effect on the WCET. Again they are not safe to be used on their own.

In Table 5.4 we can see the effects of the other reorganization optimizations. They all have a larger effect than the ones in Table 5.3. All optimizations modify the control flow graph of the program and perform basic block movements or modifications.

While the effect of block-placement is only up to two percent, it has an effect on 18 out of 33 benchmarks. We can see again that the reordering of instructions can have both a positive and negative effect on the WCET.

Codegenprepare and simplifycfg have a similar effect on the WCET and upon inspection of the code and documentation, their goals appear to be similar. While most effects are a reduction of the WCET, there are some cases where it is increased, suggesting that the impact the WCET or they make it harder for the tool to obtain a tight WCET bound.

As we can see again reordering existing instructions is not safe but may also have a positive effect.

## Assignment Placement

As the the transformations for Assignment Placement (AP), Partial Dead Code Elimination (PDCE) and Assignment Motion (AM) are source to source we are in a better position of being able to try and explain why a particular WCET effect is happening. As a reminder, Assignment Placement is the combination of Partial Dead Code Elimination and Assignment Motion (see section 3.3 for details).

The full results are displayed in Table 5.5, overall we can see that the WCET of AP is usually the worse of PDCE and AM, or even worse than both (fft1 and minver). This suggests that the improvements achieved by one optimization can be made moot by further transformations of the other optimization (usually code sinking/hoisting is done as a last step in the optimizations). This furthermore emphasises the point that uninformed control flow changes are not safe.

I will now examine the effect on all benchmarks in detail, giving examples of why the WCET increased/decreased and giving possible explanations.

- *bsort100.c:*

  The first interesting difference is this loop:

  ```
  1: for (Index = 1; Index <= NUMELEMS - 1; Index++)
  ```

  is being transformed to:

  ```
  1: __temp2__ = 100 - 1;
  2: while(i <= __temp2__)
  ```

  this change, creating a temporary variable for the number of loop iterations, could be the cause of the observed degradation in worst case performance. The other effect of the

|  | Optimizations |  |  |  |
|---|---|---|---|---|
| Benchmarks | mergefunc | mergereturn | reassociate | sink |
| bs.c | 1.000 | 1.000 | 1.000 | 1.000 |
| bsort100.c | 1.000 | 1.000 | 1.000 | 1.000 |
| cnt.c | 1.000 | 1.000 | 1.000 | 1.000 |
| compress.c | 1.000 | 1.000 | 1.000 | 1.000 |
| coop.c | 1.000 | 1.000 | 1.000 | 1.000 |
| cover.c | 1.000 | 1.000 | 1.000 | 1.000 |
| crc.c | 1.000 | 1.000 | 1.000 | 1.000 |
| duff.c | 1.000 | 1.000 | 1.000 | 1.000 |
| edn.c | 1.000 | 1.000 | 1.000 | 1.000 |
| fac.c | 1.000 | 1.000 | 1.000 | 1.000 |
| fdct.c | 1.000 | 1.000 | 1.000 | 1.000 |
| fft1.c | 1.014 | 1.012 | 1.000 | 1.002 |
| fibcall.c | 1.000 | 1.000 | 1.000 | 1.000 |
| insertsort.c | 1.000 | 1.000 | 1.000 | 1.000 |
| janne_complex.c | 1.000 | 1.000 | 1.000 | 1.000 |
| jfdctint.c | 1.000 | 1.000 | 1.000 | 1.000 |
| lcdnum.c | 1.000 | 1.000 | 1.000 | 1.000 |
| lms.c | 1.012 | 1.012 | 1.000 | 1.006 |
| loop3.c | 1.000 | 1.000 | 1.000 | 1.000 |
| ludcmp.c | 1.003 | 1.011 | 1.075 | 1.019 |
| matmult.c | 1.000 | 1.000 | 1.000 | 1.000 |
| matmult_32_32.c | 1.000 | 1.000 | 1.000 | 1.000 |
| matmult_128_128.c | 1.000 | 1.000 | 1.000 | 1.000 |
| minmax.c | 1.000 | 1.000 | 1.000 | 1.000 |
| minver.c | 1.011 | 1.010 | 1.000 | 1.102 |
| ndes.c | 1.000 | 1.000 | 1.000 | 0.997 |
| ns.c | 1.000 | 1.000 | 1.000 | 1.000 |
| nsichneu.c | 1.000 | 1.000 | 1.000 | 1.000 |
| qsort-exam.c | 1.046 | 1.000 | 1.000 | 1.000 |
| qurt.c | 1.016 | 1.013 | 1.000 | 1.055 |
| select.c | 1.000 | 1.000 | 1.000 | 1.000 |
| sin_func.c | 1.018 | 1.014 | 1.000 | 0.999 |
| sqrt.c | 1.013 | 1.014 | 1.000 | 1.012 |

**Table 5.3:** Relative WCET values for code reorganization optimizations 1

| | Optimizations | | |
|---|---|---|---|
| Benchmarks | block-placement | codegenprepare | simplifycfg |
| bs.c | 1.023 | 1.000 | 1.000 |
| bsort100.c | 0.999 | 1.044 | 1.044 |
| cnt.c | 0.992 | 1.000 | 0.923 |
| compress.c | 1.000 | 0.952 | 0.953 |
| coop.c | 1.003 | 1.010 | 1.011 |
| cover.c | 1.000 | 1.000 | 1.000 |
| crc.c | 0.976 | 0.988 | 1.065 |
| duff.c | 1.000 | 1.000 | 1.000 |
| edn.c | 1.000 | 0.999 | 1.000 |
| fac.c | 0.985 | 1.000 | 1.000 |
| fdct.c | 1.000 | 1.000 | 1.000 |
| fft1.c | 1.004 | 0.998 | 0.959 |
| fibcall.c | 1.000 | 1.000 | 1.044 |
| insertsort.c | 1.000 | 1.000 | 1.000 |
| janne_complex.c | 1.020 | 0.970 | 0.995 |
| jfdctint.c | 1.000 | 1.000 | 1.000 |
| lcdnum.c | 0.982 | 1.000 | 1.000 |
| lms.c | 1.004 | 0.995 | 0.975 |
| loop3.c | 1.000 | 0.895 | 0.895 |
| ludcmp.c | 1.005 | 1.000 | 1.000 |
| matmult.c | 1.000 | 1.000 | 1.000 |
| matmult_32_32.c | 1.000 | 1.000 | 1.000 |
| matmult_128_128.c | 1.000 | 1.000 | 1.000 |
| minmax.c | 1.016 | 1.000 | 1.000 |
| minver.c | 1.003 | 0.995 | 0.968 |
| ndes.c | 1.001 | 0.971 | 0.975 |
| ns.c | 1.000 | 1.000 | 1.000 |
| nsichneu.c | 1.000 | 0.976 | 0.976 |
| qsort-exam.c | 0.994 | 0.907 | 0.912 |
| qurt.c | 1.004 | 1.000 | 0.961 |
| select.c | 0.995 | 0.986 | 0.990 |
| sin_func.c | 1.000 | 1.000 | 0.968 |
| sqrt.c | 1.002 | 0.998 | 0.962 |

**Table 5.4:** Relative WCET values for code reorganization optimizations 2

```
1: if (Array[Index] > Array[__temp4__]) {
2:     Temp = Array[Index];
3:     Array[Index] = Array[__temp4__];
4:     Array[__temp4__] = Temp;
5:     Sorted = 0;
6: }
```

**Listing 5.1:** Snippet as transformed by PDCE

```
1: if (Array[Index] > Array[__temp4__]) {
2:     Temp = Array[Index];
3:     Array[Index] = Array[__temp4__];
4:     Sorted = 0;
5:     Array[__temp4__] = Temp;
6: }
```

**Listing 5.2:** Snippet as transformed by AM/AP

transformation is the storage of the array indexing at position *Index* + *1* in a new temporary variable, thereby only calculating the index only once not twice.

But another point is quite interesting, the PDCE and AM/AP programs only differ in the ordering of one statement, that actually makes the PDCE case 1.4% worse. See the two code listings 5.1 and 5.2.

As can be seen the *Sorted = 0;* statement is on line 5 and not on line 4 in the PDCE case. This is the only difference of the two programs, this shows that the WCET can be very sensitive to the ordering of statements and other minor changes. I suppose this is the case because of branch miss-prediction and the PDCE has to unload more information than the AM case.

- *coop.c:*

  This benchmarks contains deliberately placed dead code (for testing WCET analysis tools), and therefore the improvements are not that interesting to study.

- *fdct.c:*

  In this benchmark, no loop invariant code could be moved out of a loop and most temporary variable assignments (for expressions), are done for constants, e.g. for the following:

  ```
  __temp73__ = 13 + 2 + 3;
  ```

  which is probably more expensive than if the constant is evaluated during runtime. This would explain why the performance degraded for all tree optimizations. A smarter implementation of them could solve that problem.

```
while (a[j] < a[j - 1]) {
    cnt2++;
    temp = a[j];
    a[j] = a[j - 1];
    a[j - 1] = temp;
    j--;
}
```

**Listing 5.3:** Snippet in the original benchmark

```
__temp1__ = j - 1;
while(a[j] < a[__temp1__]){
    cnt2 = cnt2 + 1;
    temp = a[j];
    a[j] = a[__temp1__];
    a[__temp1__] = temp;
    j = __temp1__;
    __temp1__ = j - 1;
}
```

**Listing 5.4:** Snippet as transformed by AP

- *fft1.c:*

  There are two independent effects in this program. First a temporary storage for the decrement of two index variables. This should not have a large effect on the WCET, if any at all. But the second transformation does have quite a significant impact. It moves some invariant calculations out of loops of a function for calculation of the sinus. To study this effect further I have created the custom benchmark *sin_func.c*, which we will also investigate in detail later.

- *insertsort.c:*

  This is a quite interesting case, because it is actually pretty unintuitive. What the optimizations do is that they replace an expression by a temporary variable, thereby removing the expression four times. Listing 5.3 shows the original code and listing 5.4 show the code transformed by AP.

  This shows that such trivial expressions are not worth to be extracted and stored in a temporary variable. While it would be intuitive that such a temporary variable would make sense, it actually does not and the performance degrades by 9% as a result, which is quite a heavy penalty. This is something that happens in multiple benchmarks.

- *lms.c:*

Like the benchmark *fft1.c* this benchmark also contains the same custom function for the sinus calculation, and like it the WCET improvement is largely due to the move of some invariant code out of loops. See the entry for the *sin_func.c* benchmark for details about the optimization of the sinus function. Another change is also the creation of a temporary variable for storage of an indexing variable calculation. But this is not expected to have a large effect, and as we can see from the *insertsort.c* benchmark, will probably have a negative impact on the WCET.

- *ludcmp.c:*

  Again the only effect is the storage of a loop indexing calculation in a temporary variable results in the performance degradation. See the *insertsort.c* benchmark for an example of this kind.

- *minver.c:*

  This benchmark contains some intentional dead code, therefore by removing it we improved the WCET. Except for this, this benchmark is of no further interest for our considerations.

- *qsort-exam.c:*

  For this benchmark the only transformation effects are a reordering of the control flow, which results in a degradation of the performance. Again cache disruption effects is the most likely reason to explain this effect.

- *qurt.c:*

  The PDCE optimization moves an assignment to a later position in the code, this probably has some positive effect on the cache and therefore improves the performance. AP does not have this move as AM (and therefore code hoisting) is the last phase in it, this hoists the previously sunken code again to the top and the positive effect is not seen.

  This is again an indication that control flow reordering can have an effect, both positive and negative, thus making it an interesting thing to consider, but also making it not safe.

- *select.c:*

  Again we have the case where an index calculation is stored in a temporary variable (see the *insertsort.c* benchmark). The difference in the three optimizations is due to a different ordering of the control flow, but there are no actual differences in the instructions executed.

- *sin_func.c:*

  As this code fragment is contained within a couple of the used benchmarks, it is interesting to study it in detail. There is one main loop (listing 5.5) where some invariant code could be removed from (listing 5.6), this is done in all three optimizations.

  This is where all of the performance is won. The differences between the optimizations comes from two loops before where AP/AM perform transformations and PDCE does not. See the following two snippets:

```
while (fabs(diff) >= 0.00001) {
    diff = (diff * (-(rad * rad))) / ((2.0 * inc) * (2.0 * inc + 1.0));
    app = app + diff;
    inc++;
}
```

**Listing 5.5:** Original code snippet

```
__temp5__ = (rad * rad);
while((fabs(diff)) >= 0.00001){
    __temp7__ = 2.0 * inc;
    diff = ((diff * -__temp5__) / (__temp7__ * (__temp7__ + 1.0)));
    app = (app + diff);
    inc = inc + 1;
}
```

**Listing 5.6:** Snippet as transformed by PDCE/AM/AP

```
while (rad > 2 * 3.14159265358979323846)
    rad -= 2 * 3.14159265358979323846;
```

**Listing 5.7:** Original code snippet

While PDCE changes nothing in the original code (listing 5.7) AM and AP change it the following way (see listing 5.8). Even though AP/AM are able to move the calculation of *2 * PI* out of the loop this obviously decreases the performance. This is a non intuitive and non trivial result, as now distinguishing between cases where such a transformation can increase the performance becomes harder.

## Optimizations with Impact on the WCET

This section now discusses the effects of the optimizations with the strongest impact on the WCET.

```
__temp0__ = 2 * 3.14159265358979323846;
while(rad > __temp0__)
    rad = rad - __temp0__;
```

**Listing 5.8:** Transformed by AP/AM code snippet

| | Optimizations | | |
|---|---|---|---|
| Benchmarks | AP | PDCE | AM |
| bs.c | 1.000 | 1.000 | 1.000 |
| bsort100.c | 1.149 | 1.163 | 1.149 |
| cnt.c | 1.000 | 1.000 | 1.000 |
| compress.c | 1.000 | 1.000 | 1.000 |
| coop.c | 0.980 | 0.980 | 0.980 |
| cover.c | 1.000 | 1.000 | 1.000 |
| crc.c | 1.000 | 1.000 | 1.000 |
| duff.c | 1.000 | 1.000 | 1.000 |
| edn.c | 1.000 | 1.000 | 1.000 |
| fac.c | 1.000 | 1.000 | 1.000 |
| fdct.c | 1.104 | 1.103 | 1.109 |
| fft1.c | 0.928 | 0.903 | 0.927 |
| fibcall.c | 1.000 | 1.000 | 1.000 |
| insertsort.c | 1.090 | 1.090 | 1.097 |
| janne_complex.c | 1.000 | 1.000 | 1.000 |
| jfdctint.c | 1.000 | 1.000 | 1.000 |
| lcdnum.c | 1.000 | 1.000 | 1.000 |
| lms.c | 0.988 | 0.979 | 0.988 |
| loop3.c | 1.000 | 1.000 | 1.000 |
| ludcmp.c | 1.004 | 1.004 | 1.004 |
| matmult.c | 1.000 | 1.000 | 1.000 |
| matmult_32_32.c | 1.000 | 1.000 | 1.000 |
| matmult_128_128.c | 1.000 | 1.000 | 1.000 |
| minmax.c | 1.000 | 1.000 | 1.000 |
| minver.c | 0.998 | 0.998 | 1.000 |
| ndes.c | 1.000 | 1.000 | 1.000 |
| ns.c | 1.000 | 1.000 | 1.000 |
| nsichneu.c | 1.000 | 1.000 | 1.000 |
| qsort-exam.c | 1.006 | 1.003 | 1.006 |
| qurt.c | 1.000 | 0.997 | 1.000 |
| select.c | 1.018 | 1.021 | 1.017 |
| sin_func.c | 0.940 | 0.904 | 0.940 |
| sqrt.c | 1.000 | 1.000 | 1.000 |

**Table 5.5:** Relative WCET values for Assignment Placement optimizations

As the optimization *early-cse* does something similar to Assignment Motion, we can expect the same problems. And indeed Common Subexpression Elimination is not a safe optimization, though overall it seems like the gains are higher than the losses (12 improvements vs. 9 deteriorations). The most obvious reason of why the WCET deteriorates, seems to be that it is cheaper to calculate the expression again, than to retrieve the result from a temporary variable.

*Instcombine* also has quite some impact onto the WCET. As it also performs canonization and reordering of instructions, the most obvious reason of why the WCET deteriorates, is that the instructions have been reordered in such a way that the cache is disturbed.

The effects of *mem2reg* are very strong. On the one hand it manages to improve the WCET by 55% (*fdct.c*), on the other it deteriorates it by 37.8% (*cnt.c*). It obviously is not a safe optimization, but the gains that can be obtained by it are very large and therefore would be very desirable to use. It could be interesting to develop a method for predicting whether *mem2reg* has a positive impact on the program or not.

Lastly function inlining is the only optimization that never deteriorates the WCET. So it is safe to use on hard real time systems. Its only drawback is that it will increase the code size of the program.

## 5.3 Discussion of Results

This chapter will discuss some of the overarching results of the evaluation and compare them to the results reported in related work.

### Code Motion

The results of the evaluations in section 5.1 suggest that (speculative) code motions are not advisable in general. They can have an impact on the WCET, both positive and negative. This effect could be explained in a couple of ways.

- The new ordering of instructions is harder to analyse and the resulting estimation is not as tight.
- The moving of some code disrupts/enables the cache and therefore the WCET can decrease (see section 2.1 for an explanation about timing anomalies).

The effect of instruction placement on the cache performance has been reported previously [6]. From this we can conclude that (speculative) code motion (either hoisting or sinking) is by itself not a safe transformation. Li et al. [34] also report similar findings in regards to some optimizations that affect the I-cache. They ran the analysis again this time assuming a perfect cache and saw the differences disappear.

### Function Inlining

The results indicate that function inlining is an optimizations that is safe to perform. In our experiments we never encountered a case where it increases the WCET of a benchmark program.

| | Optimizations | | | |
|---|---|---|---|---|
| Benchmarks | early-cse | inline | instcombine | mem2reg |
| bs.c | 1.004 | 1.000 | 0.992 | 1.351 |
| bsort100.c | 0.926 | 1.000 | 1.000 | 1.001 |
| cnt.c | 0.923 | 0.872 | 1.002 | 1.378 |
| compress.c | 0.993 | 0.936 | 0.972 | 0.930 |
| coop.c | 1.004 | 0.995 | 1.038 | 1.098 |
| cover.c | 1.000 | 1.000 | 1.000 | 1.260 |
| crc.c | 0.971 | 0.879 | 1.085 | 1.236 |
| duff.c | 1.032 | 0.995 | 1.031 | 1.124 |
| edn.c | 0.986 | 1.000 | 0.987 | 1.016 |
| fac.c | 1.000 | 1.000 | 1.000 | 1.082 |
| fdct.c | 1.000 | 1.000 | 0.898 | 0.449 |
| fft1.c | 0.957 | 0.965 | 0.958 | 0.921 |
| fibcall.c | 1.022 | 0.982 | 0.915 | 1.172 |
| insertsort.c | 1.000 | 1.000 | 1.000 | 0.793 |
| janne_complex.c | 1.026 | 1.000 | 1.066 | 1.203 |
| jfdctint.c | 1.000 | 1.000 | 0.955 | 0.727 |
| lcdnum.c | 0.954 | 0.790 | 0.936 | 0.907 |
| lms.c | 0.973 | 0.598 | 0.969 | 0.963 |
| loop3.c | 1.000 | 1.000 | 0.998 | 0.966 |
| ludcmp.c | 1.001 | 0.998 | 1.002 | 1.000 |
| matmult.c | 1.000 | 0.973 | 1.001 | 1.001 |
| matmult_32_32.c | 1.000 | 1.000 | 1.001 | 1.001 |
| matmult_128_128.c | 1.000 | 1.000 | 1.000 | 1.000 |
| minmax.c | 1.000 | 0.623 | 0.901 | 0.848 |
| minver.c | 0.968 | 0.975 | 0.962 | 1.000 |
| ndes.c | 1.023 | 0.875 | 0.996 | 0.928 |
| ns.c | 1.068 | 1.000 | 1.004 | 1.127 |
| nsichneu.c | 1.000 | 1.000 | 0.994 | 0.917 |
| qsort-exam.c | 1.013 | 1.000 | 1.008 | 0.992 |
| qurt.c | 0.959 | 0.991 | 0.959 | 0.971 |
| select.c | 1.004 | 1.000 | 1.000 | 1.068 |
| sin_func.c | 0.968 | 0.973 | 0.969 | 0.914 |
| sqrt.c | 0.962 | 0.983 | 0.942 | 0.982 |

**Table 5.6:** Relative WCET values for optimizations with impact on the WCET

It is actually the only optimization that showed this behaviour. This is consistent with the results obtained by Davidson et al. [9]. They showed that for certain processors, function inlining was able to improve the execution time of all of their benchmark programs (or at least not make them worse).

In contrast Schwarzer [50] obtained a different result. Their WCET estimate after function inlining is always worse than the unoptimised case. Why this is the case is hard to explain. One point that Davidson et al. [9] were able to demonstrate, was that the magnitude of effect function inlining has, is dependent on the processor/compiler. So it could very well be that due to the different processor architecture used by Schwarzer, the results obtained by their experiments differ.

## General Remarks

Another interesting result is than reduction in the number of expressions to be computed, as with the optimizations Assignment Motion, Partial Dead Code Elimination, Assignment Placement and Common Subexpression Elimination (CSE), does not necessarily decrease the WCET.

This can clearly be seen with the *sin_func.c* benchmark in the case of Assignment Motion. A similar result can be seen with CSE, where some benchmarks show that the elimination of expressions results in a degradation of the WCET. The work of Lokuciejewski et al. [38] also shows a similar result, they consider Common Subexpression Elimination and Dead Code Elimination for superblocks, but also show that standard CSE/DCE (i.e. not on superblocks) can degrade the WCET. The work of Li et al. [34] also confirms this for CSE. They produce results that show a possible degradation of the WCET of 1-2% with CSE. Schwarzer [50] shows that CSE can increase the WCET by up to nearly 35%, a result we could not confirm in our experiments.

Moreover as we can see in the AP transformations of the *insertsort.c* benchmark, trivial expressions like *i - 1* are not expensive enough to replace with a temporary variable. Even if they are used four times in a loop.

## Comparison to Related Work

Comparing the results presented in this work with those of Li et al. [34] is not straightforward as the methodology differs. Their approach is to compare the WCET of the benchmark compiled with the optimization level O1 (which contains several optimizations) with the WCET of the benchmark compiled with optimization level O1 without the optimization under investigation and derive an improvement percentage from that. But nevertheless some interesting points can be made.

Firstly, except for the degradations with CSE mentioned in the last section they obtain much better performance improvements with it. This is likely due to the difference in methodology. This suggests that other optimizations unlock potential for further optimization with CSE. Furthermore their results on the optimization *simlifycfg* are similar with most changes around 5%, both in terms of improvement and degradation of the WCET. Similarly their results on *licm*, *reassociate* and most other optimizations are comparable with finding that most benchmarks cannot benefit from them. They also conclude that *mem2reg* can have a strong positive effect on the WCET, which is also something that is confirmed by this work.

The methodology of Schwarzer [50] is more comparable to the methodology of this work and for the most part the results are comparable. The differences they found was that Loop Unrolling does have a major impact on the benchmarks, but it seems to be very dependent on the processor. The other main difference is that their results differ very much for Function Inlining, but that has already been discussed in section 5.3.

CHAPTER 6

# Conclusion

## 6.1 Summary

This work examined the effects of multiple traditional compiler optimization (36 in total) on the WCET of a series of benchmarks. Most optimizations were already available in the compiler LLVM, three optimizations were implemented additionally. All optimizations were evaluated on 33 benchmarks, taken from the 2014 WCET Tool Challenge and the Mälardalen benchmarks. We computed relative WCET values (dividing the WCET of the optimized program by the WCET of the unoptimized program) for each optimization on each benchmark. From this data we were able to draw some conclusions about the effect of those optimization on the WCET.

The main results were that any form of code motion is potentially not safe as the different code layout could disrupt the cache and therefore increase the WCET, but improvements are also possible.

We also discovered that trivial expressions are not worthy of being optimized by replacing them with a temporary variable (as done by Common Subexpression Elimination and Assignment Motion).

For the optimization *mem2reg* we obtained mixed results, while the WCET improvements were larger than for any other optimization. It could also degrade the WCET by a larger amount than any other optimization.

Function Inlining was the only optimization that was able to improve the WCET (or at least not worsen it) for all benchmarks.

## 6.2 Risks

Due to our approach there are a couple of risks regarding the accuracy of the obtained results.

As we can not measure the real WCET with our selected tools, there is always the possibility that whenever we see a change in the WCET, it could be an estimation artefact and not a real effect.

Furthermore we do not know how representative the benchmarks are for the optimizations. Especially regarding AM and PDCE as they are only implemented on scalar variables and not for arrays. But most benchmarks contain array variables.

## 6.3 Future Work

Future work could focus on finding predictions/metrics of when expressions are worth to be eliminated and replaced by a temporary variable. This would be of general interest as unnecessary temporary variables make the process of register allocation harder.

Also warranting further investigation is the question of estimation artefacts. For selected benchmarks and optimizations the WCET could be measured to gauge how large the impact of estimation artefacts is.

Bordering on that lies the question, does the counterintuitive degradation of the WCET witnessed for optimizations that do not increase the number of instructions (e.g. AM, PDCE, code sinking) also present in average case execution timing. This could provide clues with regards to whether some of the obtained results are due to estimation artefacts or real results.

Furthermore we feel it would be of interest to collect more data on the behaviour of classical program optimizations on the WCET by increasing the number of benchmark programs. With more benchmarks more quantitative measurements could be made and maybe even metrics devised with which one could predict if an optimization is beneficial to the WCET.

APPENDIX A

# Benchmarks

Following is a list of all used benchmark programs, their source and a short description of each.

- *Mälardalen Benchmarks*, for the sources see:[1]
  - bs.c: Binary search for the array of 15 integer elements.
  - bsort100.c: Bubblesort program.
  - cnt.c: Counts non-negative numbers in a matrix.
  - compress.c: Data compression program.
  - cover.c: Program for testing many paths, with many switch statements.
  - 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. A lot of vector multiplications and array handling.
  - fac.c: Calculates the faculty function, with self recursion.
  - fdct.c: Fast Discrete Cosine Transform, a lot of arrays.
  - fft1.c: 1024-point Fast Fourier Transform using the Cooly-Turkey algorithm. A lot of floating point calculations.
  - fibcall: 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.

[1] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

  - lms.c: LMS adaptive signal enhancement. The input signal is a sine wave with added white noise.
  - loop3.c: A lot of loop calculations.
  - ludcmp.c: LU decomposition algorithm.
  - matmult.c: Matrix multiplication of two 20x20 matrices.
  - mimax.c: Function for calculating the min and max of two arguments.
  - minver.c: Inversion of floating point matrix.
  - ndes.c: Complex embedded code. Bit manipulation and shifts.
  - ns.c: Search in a multi-dimensional array.
  - nsichneu.c: Simulation of an extended Petri Net.
  - qsort-exam: Non-recursive version of quick sort algorithm.
  - qurt.c: Root computation of quadratic equations.
  - select: A function to select the Nth largest number in a floating point array.
  - sin_func.c: Sinus function encountered in a couple of the other benchmarks.
  - sqrt.c: Square root function implemented by Taylor series.

- *WCET Tool Challenge 2014 Benchmarks*, for the sources see:[2]

  - coop.c: A benchmark for challenging WCET analysis tools.
  - matmult_32_32.c: Matrix multiplication of two 32x32 matrices.
  - matmult_128_128.c: Matrix multiplication of two 128x128 matrices.

[2] http://www.irit.fr/wiki/doku.php?id=wtc:benchmarks:start

APPENDIX B

# Listings

# List of Tables

5.1 Statistics for optimizations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2 Relative WCET values for Loop Optimizations . . . . . . . . . . . . . . . . . . . . 30
5.3 Relative WCET values for code reorganization optimizations 1 . . . . . . . . . . . 32
5.4 Relative WCET values for code reorganization optimizations 2 . . . . . . . . . . . 33
5.5 Relative WCET values for Assignment Placement optimizations . . . . . . . . . . 38
5.6 Relative WCET values for optimizations with impact on the WCET . . . . . . . . 40

# List of Figures

4.1 Process of the AM and PDCE implementations . . . . . . . . . . . . . . . . . . . 24

# List of Listings

5.1 Snippet as transformed by PDCE . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.2 Snippet as transformed by AM/AP . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.3 Snippet in the original benchmark . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.4 Snippet as transformed by AP . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.5 Original code snippet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.6 Snippet as transformed by PDCE/AM/AP . . . . . . . . . . . . . . . . . . . . . 37
5.7 Original code snippet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.8 Transformed by AP/AM code snippet . . . . . . . . . . . . . . . . . . . . . . . 37

# Bibliography

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[2] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 2007.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] J. Beirlant, Y. Goegebeur, J. Teugels, and J. Segers. *Statistics of Extremes: Theory and Applications*. John Wiley and Sons, Ltd, 2005.

[5] F. Cassez, R. Rydhof Hansen, and M.C. Olesen. What is a Timing Anomaly? In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] W.Y. Chen, P.P. Chang, T.M. Conte, and W.-M.W. Hwu. The effect of code expanding optimizations on instruction cache design. *Computers, IEEE Transactions on*, 42(9):1045–1057, Sep 1993.

[7] J. Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, pages 20–24, New York, NY, USA, 1970. ACM.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[9] J.W. Davidson and A.M. Holler. Subprogram inlining: a study of its effects on program execution time. *Software Engineering, IEEE Transactions on*, 18(2):89–102, Feb 1992.

[10] A. Ermedahl, J. Gustafsson, and B. Lisper. Deriving wcet bounds by abstract execution. In Chris Healy, editor, *Proc. 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET 2011)*. Austrian Computer Society (OCG), July 2011.

[11] H. Falk. WCET-aware register allocation based on graph coloring. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 726–731, July 2009.

[12] H. Falk and P. Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Syst.*, 46(2):251–300, October 2010.

[13] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a WCET-Aware C Compiler. In *In Proc. of "6th Intl. Workshop on WCET Analysis"*, 2006.

[14] H. Falk, N. Schmitz, and F. Schmoll. WCET-aware Register Allocation Based on Integer-Linear Programming. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 13–22, July 2011.

[15] E. Gumbel. *Statistics of Extremes*. Columbia University Press, 1958.

[16] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 57–66, Dec 2006.

[17] J. Hansen, S. A. Hissam, and G. A. Moreno. Statistical-Based WCET Estimation and Validation. In *Worst-Case Execution Time Analysis*, 2009.

[18] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE*, pages 12–21, Jun 1998.

[19] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE*, pages 12–21, Jun 1998.

[20] C.A. Healy and D.B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *Software Engineering, IEEE Transactions on*, 28(8):763–781, Aug 2002.

[21] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003.

[22] N. Holsti, J. Gustafsson, G. Bernat, C. Ballabriga, A. Bonenfant, R. Bourgade, H. Cassé, D. Cordes, A. Kadlec, R. Kirner, J. Knoop, P. Lokuciejewski, N. Merriam, M. de Michiel, A. Prantl, B. Rieder, C. Rochange, P. Sainrat, and M. Schordan. WCET 2008 – Report from the Tool Challenge 2008 – 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis. In R. Kirner, editor, *8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*, volume 8 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-237-3.

[23] K. Kennedy. Use-definition chains with applications. *Computer Languages*, 3(3):163 – 179, 1978.

[24] G.A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[25] R. Kirner, P. Puschner, and I. Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proc. IEEE Workshop on Software Tech. for Future Embedded and Ubiquitous Systems (SEUS'05*, pages 7–10, 2004.

[26] J. Knoop, L. Kovács, and J. Zwirchmayr. Symbolic loop bound computation for wcet analysis. In Edmund Clarke, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, volume 7162 of *Lecture Notes in Computer Science*, pages 227–242. Springer Berlin Heidelberg, 2012.

[27] J. Knoop and E. Mehofer. Distribution assignment placement: effective optimization of redistribution costs. *Parallel and Distributed Systems, IEEE Transactions on*, 13(6):628–647, Jun 2002.

[28] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In V. Sarkar, B.G. Ryder, and M.L. Soffa, editors, *PLDI*, pages 147–158. ACM, 1994.

[29] J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In D.W. Wall, editor, *PLDI*, pages 233–245. ACM, 1995.

[30] P. Kulkarni, S. Hines, D. Whalley, J. Hiser, J. Davidson, and D. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim*, 2, 2005.

[31] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 12–23, New York, NY, USA, 2003. ACM.

[32] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, March 2004.

[33] H. Leather, M. O'Boyle, and B. Worton. Raced profiles: efficient selection of competing compiler optimizations. *SIGPLAN Not.*, 44(7):50–59, June 2009.

[34] H. Li, I. Puaut, and E. Rohou. Traceability of flow information: Reconciling compiler optimizations and wcet estimation. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 97:97–97:106, New York, NY, USA, 2014. ACM.

[35] Y.-T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[36] P. Lokuciejewski, H. Falk, and P. Marwedel. Wcet-driven cache-based procedure positioning optimizations. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 321–330, July 2008.

[37] P. Lokuciejewski, H. Falk, P. Marwedel, and H. Theiling. Wcet-driven, code-size critical procedure cloning. In *Proceedings of the 11th International Workshop on Software &#38; Compilers for Embedded Systems*, SCOPES '08, pages 21–30, New York, NY, USA, 2008. ACM.

[38] P. Lokuciejewski, T. Kelter, and P. Marwedel. Superblock-based source code optimizations for wcet reduction. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1918–1925, June 2010.

[39] P. Lokuciejewski and P. Marwedel. Combining worst-case timing models, loop unrolling, and static loop analysis for wcet minimization. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 35–44, July 2009.

[40] P. Lokuciejewski and P. Marwedel. Combining worst-case timing models, loop unrolling, and static loop analysis for wcet minimization. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 35–44, July 2009.

[41] P. Lokuciejewski, S. Plazar, H. Falk, P. Marwedel, and L. Thiele. Approximating pareto optimal compiler optimization sequences–a trade-off between wcet, acet and code size. *Softw. Pract. Exper.*, 41(12):1437–1458, November 2011.

[42] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[43] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[44] A. Prantl, J. Knoop, R. Kirner, A. Kadlec, and M. Schordan. From trusted annotations to verified knowledge. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET 2009)*, pages 39–49, Dublin, Ireland, June 2009. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-252-6.

[45] J. Reineke and R. Sen. Sound and Efficient WCET Analysis in the Presence of Timing Anomalies. In Niklas Holsti, editor, *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–11, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[46] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In Frank Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[47] S. Schaefer, B. Scholz, S.M. Petters, and G. Heiser. Static analysis support for measurement-based wcet analysis. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, work-in-progress session*, 2006.

[48] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, and C.W. Probst. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics (OASIcs)*, pages 11–21, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[49] M. Schordan. Source-to-source analysis with satire - an example revisited. In Florian Martin, Hanne Riis Nielson, Claudio Riva, and Markus Schordan, editors, *Scalable Program Analysis*, number 08161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[50] M. Schwarzer. Untersuchung des einflusses von compiler-optimierungen auf die maximale programm-laufzeit (wcet). Master's thesis, Universität Dortmund, 2007.

[51] J.A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, Oct 1988.

[52] L. Tan. The worst case execution time tool challenge 2006: Technical report for the external test. In *In Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06*, 2006.

[53] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes, Postfach 151141, 66041 Saarbrücken, 2004.

[54] M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991.

[55] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*, pages 295–303, Sept 2005.

[56] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Measurement-based worst-case execution time analysis. In *Software Technologies for Future Embedded and Ubiquitous Systems, 2005. SEUS 2005. Third IEEE Workshop on*, pages 7–10, May 2005.

[57] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[58] M.F. Younis, T.J. Marlowe, G. Tsai, and A.D. Stoyenko. Applying compiler optimization in distributed real-time systems. Technical report, Department of Computer and Information Science, New Jersey Institute of Technology, 1995.

[59] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller. Improving wcet by optimizing worst-case paths. In *in IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 138–147, 2005.

[60] W. Zhao, P. Kulkarni, D. Whalley, C. Healy, F. Mueller, and G.-R. Uh. Timing the wcet of embedded applications. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 472–481, May 2004.