Linköping University | Department of Computer and Information Science
Bachelor's thesis, 16 ECTS | Datateknik
2022 | LIU-IDA/LITH-EX-G--2022/028--SE

Compiler optimization VS WCET

- battle of the ages

Kompilatoroptimering VS WCET

Max Nordin
Tova Harrius

Supervisor : Filip Strömbäck
Examiner : Lena Buffoni

External supervisor : Saab AB

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

Optimization by a compiler can be executed with many different methods. The defence company Saab provided us with a mission, to see if we could optimize their code with the help of the GCC compiler and its optimization flags. For this thesis we have conducted a study of the optimization flags to decrease the worst case execution time. The first step to assemble an effective base of flags was reading the documentation for the flags. We then tested the different flags and analysed them. In the end we ended up with four chosen sets that we saw fitted to be discussed and analyzed further. The results did not live up to our expectations, as we thought the flags would optimize the execution time. The flags in the majority of cases gave an, although small, increase of the execution time. We only had one set where the flags gave us a decrease, which we called the Expensive Optimization. With these results we can conclude that Saab do not need to change their existing set of optimization flags to optimize their compiler further.

Acknowledgments

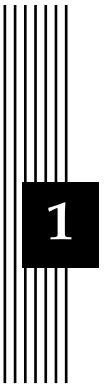
We would like to thank our examiner, Lena Buffoni, and our supervisor Filip Strömbäck for giving us guidance for the technical aspects, the presentation and this paper. We would also like to thank Saab for giving us this opportunity to make this study for them. On Saab we had three supervisors that we would like to give an extra thank you. Sara Hansson who had read our report several times to make sure that no sensitive information had been leaked, and has treated us to extra fine Selecta coffee; Niklas Pettersson who helped us with the systems, the execution of our plans and has provided a great deal of knowledge to our thesis; and lastly Erik Hansson, who has been our compiler expert on Saab and has been giving us great feedback on our thesis.

Last but not least, we would give a huge thank you to friends and family who has been supportive throughout this work.

Contents

List of Figures

List of Tables



1 Introduction

1.1 Motivation

Saab strives to minimize their usage of computational resources in different ways. Regarding the optimization of the execution time, the company has not yet fully utilized their compiler. Saab wants to know how much their compiler can be utilized in regard of the execution-time for different code tests provided by the company.

1.2 Aim

The aim for this report is to study and assemble a base of optimization flags to improve the Worst Case Execution Time, WCET. The base of optimization flags must be suitable for a flight critical software. The tests are provided by Saab, but the result and evaluation can help both companies and programmers alike to utilize their compiler optimization.

1.3 Research questions

1. How will the chosen optimization flags affect the test cases in regards to:
   - WCET?
   - The object code?
2. Are there any individual optimization flags that are of extra interest when WCET is being improved?
3. Are there any individual optimization flags that are of extra interest when WCET is being worsened?

1.4 Delimitations

This thesis was made for the Swedish aerospace and defence company Saab with their system. This means that these tests with the flags were only executed on one system, with a code standard and with one version of the GCC compiler. Thus, the flags that are used in

this thesis could result in a different behaviour if one were to try it on another system, compiler or with another code standard. The tests provided for this thesis was made by and for Saab. This means that the tests were made with regard to Saabs system, code standard and compiler.

Another delimitation for this thesis is that only the optimization flags from the different optimization groups, -O1, -O2 and -O3, were considered and tested.

Due to time constraints and the massive amount of object code acquired for this thesis, only a few chosen modules had their object code analyzed.

1.5 Restrictions

Since Saab is a company with confidential information, we will not discuss or go into all the details within the method and discussion chapter, and we will change any potential domain specific tests names that will occur. Furthermore, any potential flags given to us by Saab not relating to optimization will not be evaluated further on within this thesis.

This paper will not include every flag combination possible, since there are in total 56 flags that can be combined, which will result in 72,057,594,037,927,900 different combinations.

2 Theory

2.1 Real-time systems

Saab is a company that develop products that is using safety critical systems. For a safety critical system a hard real-time system is often used [15]. The hard real-time systems processes have to meet deadline. Hard real-time systems is used when a failure of a deadline could result in a loss of lives and/or properties, for example in air crafts and hospital equipment. In comparison, a soft real-time system should meet the deadline, but can be missed if the result of the system can still be of use after the missed deadline [9]. An example of this can be if a few bits misses the deadline but is arrived at a later stage in a video game, a lag will occur, but the game will not crash.

During development of real-time systems, it is common to disable most optimizations done by the compiler. One reason to do this is to easier map the resulting machine code, the object code, back to the source code if an error would occur while executing the code [1]. This is especially common in safety critical systems such as airplanes. The mapping of the object code is a requirement for the highest software level in the guideline DO-178B.

2.1.1 DO-178B

Saab is a company that develop and produce safety critical products, and thus follows the guideline DO-178B. DO-178B is a guideline for the software for airborne systems [1]. DO-178B or DO-178C, which is the newest version by the time this thesis is written, is the standard used by companies in the avionics industry, such as Boeing and British Airways. The guideline includes different processes and documents like planning, development, verification and requirements. These are divided into different software levels.

2.1.1.1 Software Level

Software Level or Design Assurance Level/DAL is an assignment required in DO-178B, including failure conditions that shows the severity of different faults within the software [1]. In DO-178B it is declared that each requirement in the avionic software is required to be as-

[1] Information given by Saab supervisor, Niklas Pettersson

signed a Software Level. The list below shows the failure condition categorization and its corresponding software level definitions [11].

- Catastrophic/A

  Software abnormalities that can produce a fault which in turn can result in that the aeroplane can not fly and/or land safely. For example, engines will not start.

- Hazardous/B

  Software abnormalities that can produce a fault which in turn can result in serious or fatal injury. For example, warning systems will not start.

- Major/C

  Software abnormalities that can produce a fault which in turn can result in discomfort. Can result in injuries. For example, radio will not start.

- Minor/D

  Software abnormalities that can produce a fault which in turn can result in a slight reduction of the safety marginals. For example, the flight planning system will not start.

- No effect/E

  Software abnormalities that can produce a fault which in turn results in no negative effect. For example, the coffee machine will not start.

The different software levels have different requirements that must be met. Table 2.1 illustrates an example of how a *Process Objectives and Outputs* table could look like. The table include the requirement, which chapter the requirement is referenced to in DO-178B, the different Software Levels (SW) applicability, which output the requirement generates with its reference and the different software levels. The requirements for Applicability by SW Level is what the requirements the function must fulfill and C by SW Level is the requirements to control that the right output was generated.

	Objective		Applicability by SW Level				Output		C by SW Level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
1	Requirement	X.X	●	○	○		Requirement	X.X	①	①	②	②

Table 2.1: An example of how a process objectives and outputs table could look like.

The definition for the different symbols in the table is included in table 2.2.

Legend:	
●	Fulfill the requirement with independence [2].
○	Fulfill the requirement.
Blank	The application decides if requirement is fulfilled.
①	Fulfill the requirement for Control Category 1.
②	Fulfill the requirement for Control Category 2.

Table 2.2: Legend to table 2.1

[2]The requirement of independence is fulfilled when an individual that is not the programmer is reviewing and testing the code.

Control Category 1 and Control Category 2, CC1 and CC2, are different checks that control the configuration of the software life cycle data. In the table 2.3 the different objectives associated with the two categories is shown. SCM stands for Software Configuration Management.

SCM Process Objective	Reference [1]	CC1	CC2
Configuration Identification	7.2.1	•	•
Baselines	7.2.2a,b,c,d,e	•	
Traceability	7.2.2f,g	•	•
Problem reporting	7.2.3	•	
Change control - integrity and identification	7.2.4a,b	•	•
Change Control - tracking	7.2.4c,d,e	•	
Change Review	7.2.5	•	
Configuration Status Accounting	7.2.6	•	
Retrieval	7.2.7a	•	•
Protection against Unauthorized Changes	7.2.5b(1)	•	•
Media Selection, Refreshing, Duplication	7.2.7b(2),(3),(4),c	•	
Release	7.2.7d	•	
Data Retention	7.2.7e	•	•

Table 2.3: Requirements that is included in each of the Control Categories

The higher software levels have a higher control category, which result in that they have more requirements which must be fulfilled. Parts of the requirements are to measure the WCET and WCSU for the application and to have a readable object code that ensures the wanted result. Even if an optimization flag decreases the WCET significantly, if the same flag also makes it hard to trace the object code to its source it can not be used, since it results in a requirement not being fulfilled.

2.1.2 WCET

To facilitate the scheduling the Worst Case Execution Time, WCET, is used. The scheduler uses WCET to ensures that the application will have a sufficient execution time slot in the scheduler each time the the application is initiated[3]. WCET can be acquired either by running the same test case thousands of times, using a code analyzer or calculating the execution time manually [15]. The test run that had the longest execution time becomes the WCET.

The execution time is measured in the amount of CPU-ticks, in order to generalize and to make the results anonymous.

2.1.3 WCSU

Worst Case Stack Utilization, WCSU, is measured in the amount of bytes an application might need in order to execute properly. WCSU can be acquired in almost the same way as WCET, either by running the test case once, using a code analyzer or calculating the number of bytes used manually [15]. By having a WCSU the scheduler can be consistent in its use with bytes.

2.2 The testing hardware

In this thesis two different types of hardware were used to execute the tests, *Emulated Hardware* and *Dedicated Hardware*[4].

[3]Information provided by one of our supervisors at Saab, Niklas Pettersson

[4]The information for the testing hardware is provided by one of our supervisors at Saab, Niklas Pettersson

2.2.1 Emulated Hardware

The emulated hardware is a resource that is used to test the functionality of the applications. This hardware has the same processor as the corresponding aircraft, and is cheap to use due to its ability to run on a regular PC. The emulated hardware gives a fairly accurate result.

2.2.2 Dedicated Hardware

The dedicated hardware is the same hardware as the corresponding aircraft. This hardware is more expensive to utilize than the emulated hardware, however the dedicated hardware gives an exceptionally accurate result.

2.3 Compilers

In its most basic form, a compiler is a computer program which is able to read code in one language and translate it to another without introducing or removing any functionality to the translated code. [2]. The major three steps in a compiler can be seen in figure 2.1.

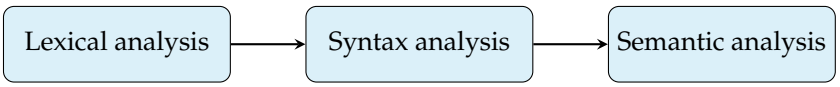


Figure 2.1: The steps in a compiler

2.3.1 Lexical analysis

Lexical analysis is the first phase a compiler performs, sometimes also called *scanning* [2]. During this phase the compiler reads the source code and group characters together into *lexemes*. With these lexemes the compiler creates *tokens* that represents the different elements of the source code. Such a token could look as follows: `<token-name, attribute-value>` where the `token-name` represent the type of token, called *symbol*, that will be represented. The `attribute-value` will hold an identification value related to the symbol, if extra information needs to be stored. Such information can be a variable name, stored value or data type. This information would be stored in a *symbol table* where the symbols entry can be accessed with use of the `attribute-value`. If no information needs to be stored in the symbol table the `attribute-value` may be omitted from the token.

Below, in figure 2.2 is a statement which is scanned and the produced tokens can be seen in figure 2.3. Information of tokens are stored within a symbol table if needed, as shown in table 2.4.

```
a = b + (c - d) * 10
```

Figure 2.2: Pseudo code expression

```
<id,1> <=> <id,2> <+> <(> <id,3> <-> <id,4> <)> <10>
```

Figure 2.3: Tokens generated from the pseudo code in figure 2.2

2.3.2 Syntax analysis

The syntax analysis or sometimes called *parsing* is the second phase of a compiler [2] where a tree structure representing the source code structure is built.

This tree structure are often a *syntax tree* or a *parse tree* and is created out of the tokens outputted from the the *lexical analysis*, specifically the *token name*.

id	name	value
1	a	...
2	b	...
3	c	...
4	d	...

Table 2.4: Symbol Table generated from the pseudo code in figure 2.2

The parse tree represents the structure of a given statement [2] and in order to construct a parse tree, a *grammar* need to exist that explains how a statement should be interpreted. A basic grammar can be seen in figure 2.4.

```
S  ⇒  id = E

E  ⇒   E + E
   ⇒   E - E
   ⇒   E * E
   ⇒    (E)
   ⇒     id
   ⇒  number
```

Figure 2.4: Syntax grammar

The syntax analyzer starts by creating a root node that represent the entire statement given to it. Guided by the grammar, the syntax analyzer then creates child nodes to the root node. Thereafter, repeating the actions and expands the tree from the child nodes accordingly. This is done until the entire statement is represented within the parse tree. Using the grammar in figure 2.4 and the tokens given from figure 2.3, the parse tree shown in figure 2.5 can be constructed.

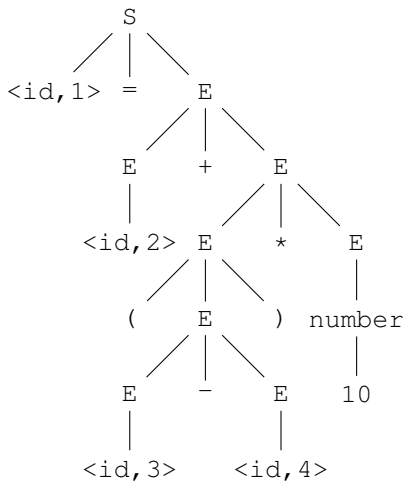


Figure 2.5: Parse tree built from the tokens in figure 2.3 with the grammar in figure 2.4

A simplified tree structure is the syntax tree and through its structure, the syntax tree shows which operations that should be performed [2] and in what order. Each node in the syntax tree represents an operation, and its children represents the arguments said operation. The syntax tree in figure 2.6 represents the same statement figure 2.2 as the parse tree in figure 2.5 represent.

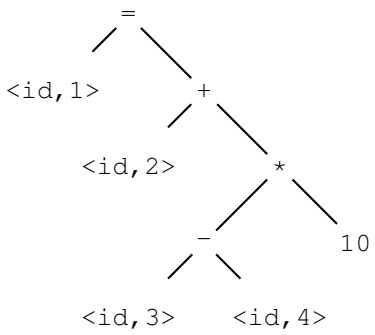


Figure 2.6: A syntax tree based on the tokens from figure 2.3

2.3.3 Semantic analysis

In the third phase, *semantic analysis*, the consistency to the language definition is checked. With the help of the *syntax tree* and the *symbol table*, the semantic analysis can verify that operands has access to the correct amount of arguments, and *type checking*. In some cases *coercion* may be performed if the language allows for it to happen. Coercion is the act of converting a *data type* into an other type in order to perform an operation, for example in order to perform arithmetic operation between an integer and floating-point number, the integer may be coerced into a floating-point number [2].

2.4 Intermediate Code Generation & Code Optimization by the Compiler

Compilers can utilize intermediate representations of the source code in order to optimize program code [2] in a multitude of aspects, such as execution time, memory utilization or code size.

2.4.1 Directed Acylic Graph

A *Directed Acylic Graph (DAG)* is an intermediate representation the compiler can utilize to easily find sections that it may be able to optimize. The DAG makes it possible to find among others, *dead code* and *common subexpressions* [2].

The DAG is an alternative to the syntax tree, the main difference is that in a DAG, a node can have multiple parents while in a standard syntax tree, a node has only one parent [2].

```
a * (b + c) - (b + c) / d
```

Figure 2.7: Mathematical Expression

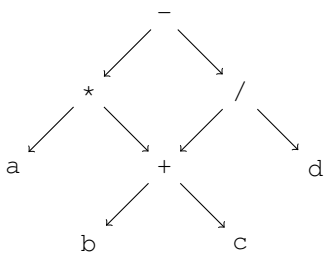


Figure 2.8: DAG of the expression from figure 2.7

2.4.2 Common Subexpression Elimination

Common subexpressions exists when identical expressions within code is repeated. A common subexpression can be detected within a DAG if a node has multiple parents [2].

Through *common subexpression elimination* the compiler can make sure that a program only need to calculate identical expressions once. This is done by storing the calculated value in a variable and reading the variable if the value is needed instead of calculating it again [5] [2].

A Common subexpressions can be seen in figure 2.9 since there exists two identical subtrees with a plus-node as root, but it is more apparent in the figure illustrating DAG, figure 2.8 which represents the same expression figure 2.7.

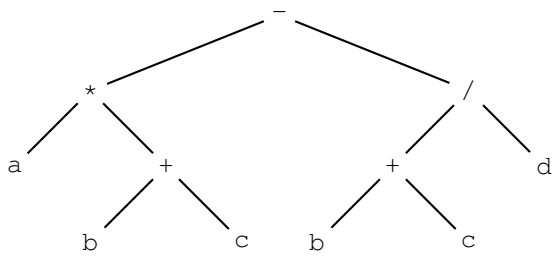


Figure 2.9: syntax tree of the expression in figure 2.7

2.5 The GCC compiler

The GNU Compiler Collection also known as GCC is a compiler for a multitude of programming languages [5]. Examples of languages that GCC support are C, C++, Fortran and Ada. GCC allows the user to optimize their code utilizing the compiler by enabling optimization flags.

2.5.1 The align flags

`-falign-functions=`*n*, `-falign-jumps=`*n*, `-falign-loops=`*n* and `-falign-labels=`*n* are four optimization flags whose goal is to align the generated code with a specified amount of bytes. The number of bytes to align by is a *power-of-two* greater or equal to *n* [8]. If no value for *n* is given, the machine default value is chosen, which often is 1, meaning no alignment will be performed.

These flags are best used together since using a subset of these increases the risk for slowdown. This is especially true for `-falign-labels` since this flag may need to insert dummy instructions [8]. These dummy instructions can then be mitigated with the help of `-falign-jumps` and `-falign-loops`.

A short description of the align flags would be:

- `-falign-functions` allow the compiler to align the start of functions to a *power-of-two* greater or equal to *n*.
- `-falign-lables` allows the alignment of lables within the code.
- `-falign-jumps` aligns branching targets.
- `-falign-loops` will align the begining of loops.

[5] Information on common subexpression elimination aquired from, https://cran.r-project.org/web/packages/rco/vignettes/opt-common-subexpr.html, (accessed: 07.12.2021)

Alignment requirements are often decided by the hardware that is used[6]. A CPU can only access 8, 16 or 32 bits at a time in the usual case. Some accesses may be performed at a faster rate depending on the chosen alignment and hardware. For example, in some x86 CPUs, alignment[7] can result in a more effective branch prediction if the branch destination is aligned at 2 or 4 bytes.

2.5.2 Common Subexpression Elimination

The optimization flags `-fcse-follow-jumps` and `fcse-skip-blocks` are both flags that enable *common subexpression elimination* [8]. Even though they both perform the same kind of optimization, they do this in different situations.

`-fcse-follow-jumps` allows the compiler to continue to scan instructions after a branch if those instructions can not be reached in any other way. An example of this could be and else clause, since the else clause can only be accessed if the corresponding if-statement fail.

`-fcse-skip-blocks` enables the compiler to follow jumping instructions which conditionally skips code blocks, and continue to scan instructions from there. An example could be an if-statement without an else clause, and where the if-segment is not entered.

2.5.3 Block reordering

A *basic block* is a continuous sequence of code with only one entry and exit point, in other words a segment without branching [2, 4]. The `-freorder-blocks` flag aims to minimize the number of branches taken by reordering these basic blocks within the code [8]. This also improves the locality of the code.

2.5.4 Expensive optimization

The creators of GCC give a vague description for the flag `-fexpensive-optimizations`. The description, which is as follows; *"Perform a number of minor optimizations that are relatively expensive"* [8], is unclear whether the flag optimize for execution time, memory usage or another resource. Fortunately *The definitive guide to GCC* [13] provides a slightly more detailed description. The description for `-fexpensive-optimizations` tells the reader that the flag optimizes in regard to processing time.

[6] Information about alignment requirement and alignment effect given at Saab, Erik Hansson
[7] Alignments effect on branch prediction given by our thesis supervisor, Filip Strömbäck

3 Method

Prior to starting the practical part of this thesis a discussion with Saab was held, regarding how the study should be performed. An agile approach to the testing was preferred. Thus it could be decided which flag should be removed or added to the different tests, without approval from Saab.

3.1 Pre-study

Before deciding on an initial set of optimization flags, a study was performed ,by the authors, of the optimization flags within GCC version 4.3. With this information a set of flags could be assembled that is more likely to decrease the WCET of the code. The GCC manual [8] provides basic information regarding the different flags. This information combined with the more detailed descriptions from William Von Hagen in his book *The definitive guide to GCC* [13], which gives the reader a good understanding regarding the optimization flags. With this knowledge, an initial set of optimization flags was created for testing.

3.2 The provided tests

This thesis was executed with domain specific tests that were provided by Saab. The tests showed the WCET and the WCSU for different applications. In this thesis the applications are numbered, leading to the use of the term *Module X*. In *Module X* the corresponding tests for an application, which is more likely to have a similar code structure, *X*, is located. One application/module could include several tests. Additionally the tests are numbered sequentially and were written with the format *Test Y* where *Y* is the tests number. This means that the *Module 1* could include test 1-2 and *Module 2* test 3-4 etc. Due to the anonymized result we will not tell which tests a module contain.

3.3 The selection of flags

The *initial set* of flags, that were first used by Saab, were used on all the provided tests each time they were executed, once by themselves and the rest of the times combined with other flags. Depending on the result it was decided if a flag were to be included in future flag bases.

If a flag gave n decrease in WCET or if WCET stayed the same the flag could be used in other test sets. If the flag did not change the WCET the flag could improve WCET in combination with other flags. If the flag gave a major increase in WCET this flag could be excluded from future tests.

Some of the studied optimization flags were of an extra interest to try and were tried with previously tested flags, or by themselves, excluding the initial set.

The different flag used for each test run can be found in appendix figure **??**, 7.6, 7.13, 7.18 and 7.21.

3.4 Testing

When a set of optimization flags had been chosen, the flags had to be added to a makefile associated to a given hardware modules test file. Thereafter, the tests are compiled for emulated hardware testing and execution. The resulting values from the *emulated hardware* testing is then analysed and a decision is made if the set should be compiled for *dedicated hardware* testing for further testing.

3.4.1 Testing on emulated hardware

The first phase of testing is execution on *emulated hardware*, which should give a fairly accurate representation of the tests in regard of WCET and WCSU. The resulting WCET and WCSU values for the module is then stored within a database.

3.4.2 Analyzing the test result

The next phase is to analyze the emulated hardware test results. In order to evaluate the different combination of optimization flags, a set of reference values need to be established. All tests are executed with the set of optimization flags utilized by Saab and are used as a reference to compare changes in WCET, WCSU and object code, henceforth it shall be known as *the original set*.

With the analyzed results one can perform an educated choice to add or remove any specific flags for future flag sets.

3.4.2.1 Result

Following the execution of a test, the resulting WCET and WCSU is stored in a database. The resources used for WCET and WCSU respectively, are processing time and memory stack usage. If one can conclude that a specific flag significantly increase the WCET for a majority of the tests, there may be no reason to continue testing that specific optimization flag. The opposite can be said if a optimization flag improves the result. Since processing time is more expensive than memory stack usage, the main goal is to decrease the WCET. This causes the values for WCSU to be of less significance for the result, but it can be used to evaluate the optimization flags and see how they are affecting the memory stack.

3.4.2.2 Object code

After acquiring and analyzing the results, the object code, for a few chosen modules, were analyzed to see the different changes within the code.

Due to lack of time and the amount of tests executed, this thesis focus on the object code corresponding to the modules *module 2*, *module 5* and *module 30*.

3.4.3 Testing on dedicated hardware

If the difference in WCET from the results of the *emulated hardware* is considered to be significant, testing on *dedicated hardware* will be performed for a few chosen *hardware modules*. The resulting values from the dedicated hardware testing is much more precise than the emulated hardware testing. But due to lack of time, we can only execute a few tests on the dedicated hardware. As before, the resulting values are stored in a database and analysed, the object code will not be analysed again since it is identical between emulated hardware and dedicated hardware.

4 Results

4.1 Test runs

In the this chapter, the results of the optimization will be presented. Every optimization method will use the same structure to present its data. We start by presenting the WCET averages of emulated hardware and then the WCET averages of dedicated hardware. Thereafter the worst and best individual tests, which can be found in the appendix, are presented and lastly the effects for the object code.

When the result of object code is presented, negative or positive branching prediction may be mentioned. A positive branching prediction will relate to a branch being likely to be taken, and a negative not likely to be taken.

4.1.1 The align flags

Studying the average change in WCET for the modules in figure 4.1, shows the result when the compiler is allowed to choose the value of n (the default value for the align flags). An increase of WCET is seen in the general case on the emulated hardware. The largest impact is in *Module 7* where an average increase with 1115 CPU-ticks or 1.14% occurred. *Module 22* has the greatest difference in percentage of all tested modules with an increase of 9.78% and an absolute change of 404 CPU-ticks.

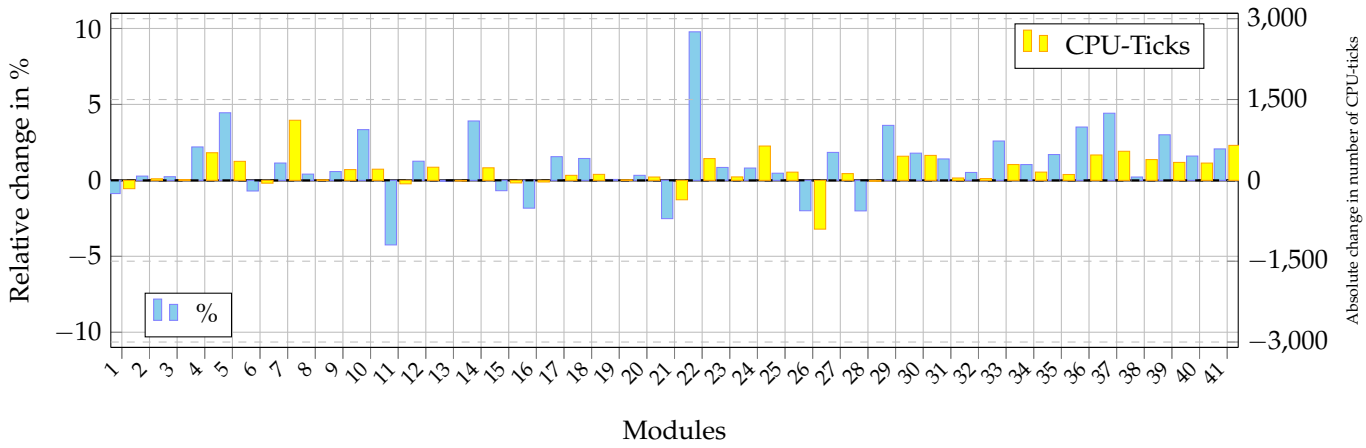


Figure 4.1: Average change in percent and CPU-ticks per module performed by the align flags on emulated hardware where the compiler chose the value of *n*

Performing the same inspection of the modules in figure 4.2 where our chosen value of *n=4*, one can find the highest CPU-tick difference in *Module 7* with an increase of 1357 CPU-ticks or 1.38%. Looking after the greatest percentile change, it can be found in *Module 11* with a improvement of 8.29% representing 125 CPU-ticks.

The highest WCET increase of a single test, produced by the align flags with the emulated hardware, ended up with an increase of 1538 CPU-ticks. This value equals an increase of 5.58% and was found in *Module 25*.

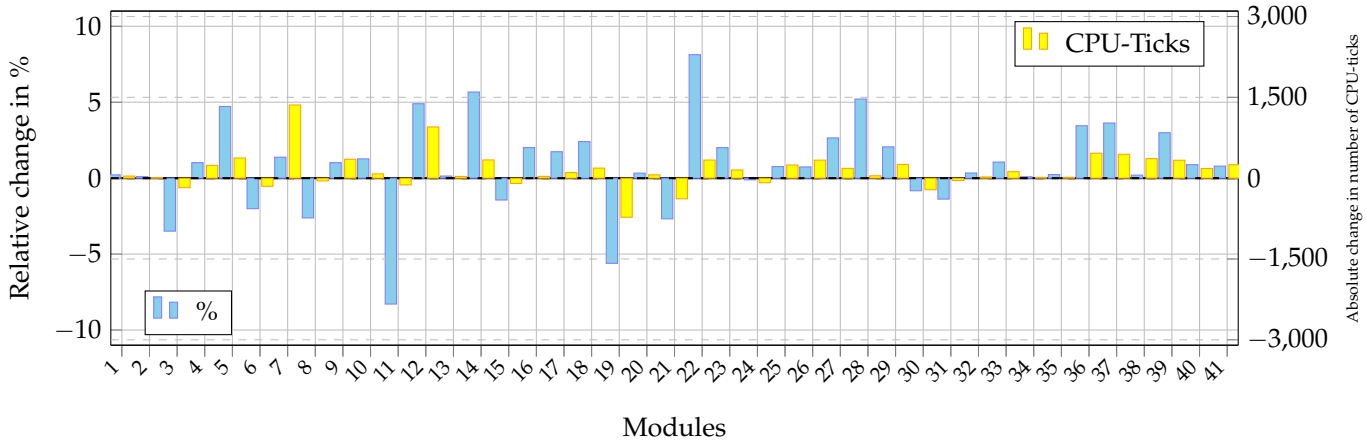


Figure 4.2: Average change in percent and CPU-ticks per module performed by the align flags on emulated hardware with chosen value of *n=4*.

The average WCET change on dedicated hardware can be seen in 4.3 where *Module 2* has increased its WCET with 21.83 CPU-ticks or 0.23%. *Module 27* has its average WCET lowered with 35.67 CPU-ticks. Which is equal to a improvement of 0.59%.

Looking at the result from the dedicated hardware in regard of single tests. The highest increase was *Module 2* which was much lower than the result from the emulated hardware. This increase ended up on 57 CPU-ticks, representing an increase of 0.0058%. The greatest decrease of a single test's WCET performed by the emulated hardware can be seen in *Module 38* with an decrease of 1122 CPU-ticks representing 0.45%.

The dedicated hardware managed to get a decrease of 89 CPU-ticks at most, which is a change of 0.0073%. This was found in *Module 27*.

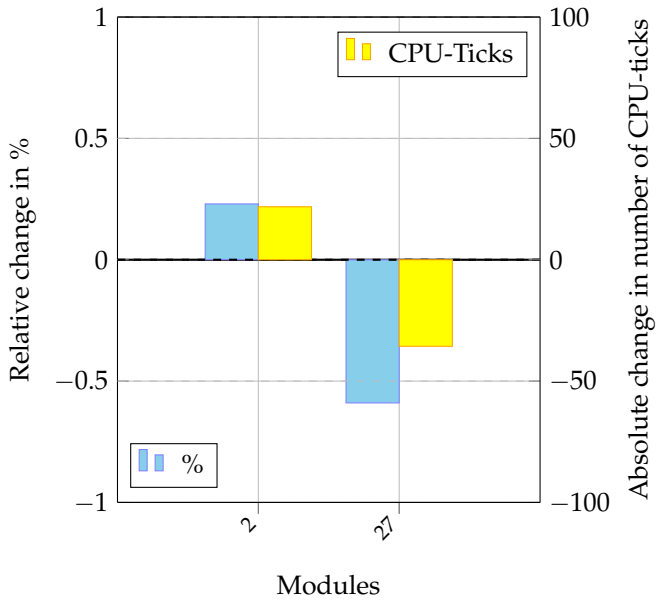


Figure 4.3: Average change in percent and CPU-ticks per module performed by the align flags on dedicated hardware.

In total we saw an increase of 24613 CPU-ticks with the emulated hardware, which equals to an increase of 0.51%.

By observing the object code, the *align flags* resulted in an identical code compared to the *original set* which does not utilize any alignment flags. There was no difference in the code if the compiler decided what the input value n for the flags should be, or if the value n was manually set to 4.

4.1.2 Common Subexpression Elimination

The result of the averages for emulated hardware can be seen in 4.4. Almost all modules had an worsening of WCET as an effect of common subexpression elimination, with *Module 7* being the one with the greatest change of 1837 CPU-ticks. This equals to a change of 1.87%. The largest change in percent can be found in *Module 5*, having an increase of 5.95% or 471.67 CPU-ticks.

When looking at single test results from the effect of *Common Subexpression Elimination* on emulated hardware, both the largest increase and decrease of WCET was found within *Module 38*. This resulted in a WCET increase of 2391 CPU-ticks and a decrease of 4845 CPU-ticks and relates to a increase of 1.00% and a decrease of 1.97%.

The dedicated hardwares average WCET change seen in 4.5 show that both *Module 2* and *Module 27* had their WCET improved. *Module 2* had its CPU-ticks decreased with 13.52 which relatets to a change of 0.10%. Looking at *Module 27*, an decrease of 38.67 CPU-ticks or 0.47% are observed.

The result from a single test from the dedicated hardware has a much lower impact than the emulated hardware with an increase of 35 CPU-ticks and a decrease of 127 CPU-ticks. Representing an increase of 0.0039% and a decrease of 0.0097% and relates to *Module 27*.

With the emulated hardware, common Subexpression Elimination ended up giving an total increase of WCET with 21057 CPU-ticks or 0.44%.

The optimization flags regarding to common subexpression elimination did not have a major impact on the object code. In fact, the code is almost identical to the object code of the *original set*. The changes that was shared by *Module 2*, *Module 5* and *Module 30* was elimination

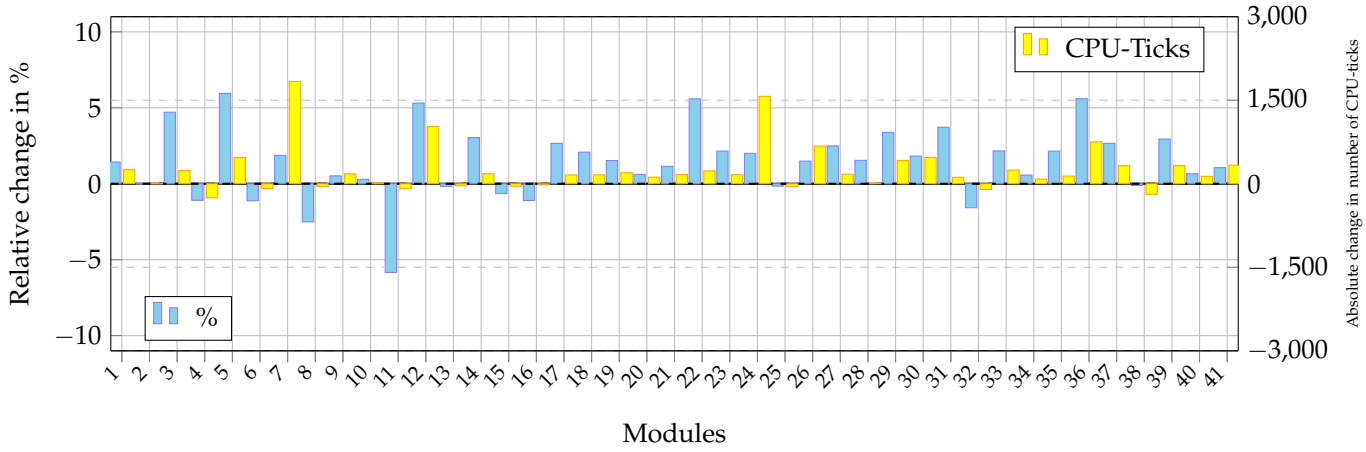


Figure 4.4: Average change in percent and CPU-ticks per module performed by Common Subexpression Elimination on emulated hardware.

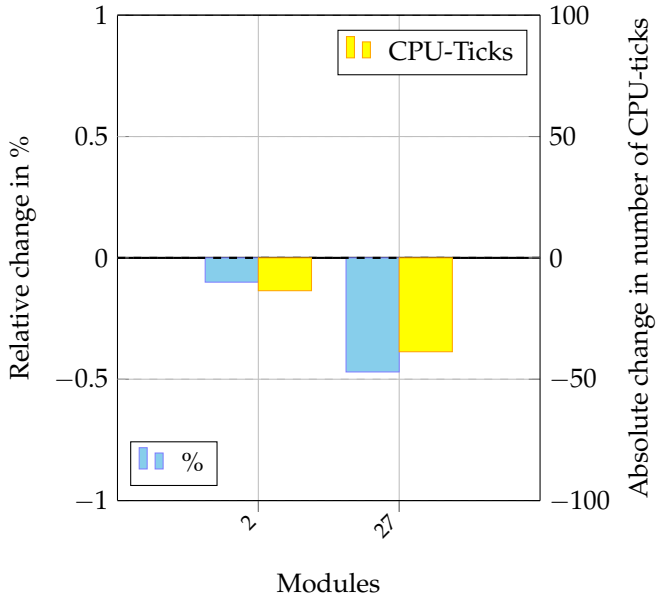


Figure 4.5: Average change in percent and CPU-ticks per module performed by the Common subexpression Elimination on dedicated hardware.

of a few load instructions and register move instructions. There was also some cases where store instructions had changed into register move instructions. The same can be said about a very small amount of addition instructions as they had been changed to register move. The final difference was a change of positive branch prediction to negative branch prediction in many cases.

4.1.3 Block reordering

Allowing the compiler to reorder the basic blocks of the code resulted in an clear increase of WCET compared to the *original set* as seen in figure 4.6. Out of 41 modules, only five had their WCET improved by block reordering when looking at the averages of each module, and none of these five had any major change in regards of WCET. The greatest difference of CPU-ticks is seen in *Module 38* with a value of 14921, representing a increase of WCET with

8.42%. This change is extreme in relation to all other averages within this set. The second largest difference of 2359 CPU-ticks is found in *Module 7* which is equal to a worsening of 2.41%. When studying *Module 22*, the greatest percentile change was found with a value of 9.44% or 390 CPU-ticks.

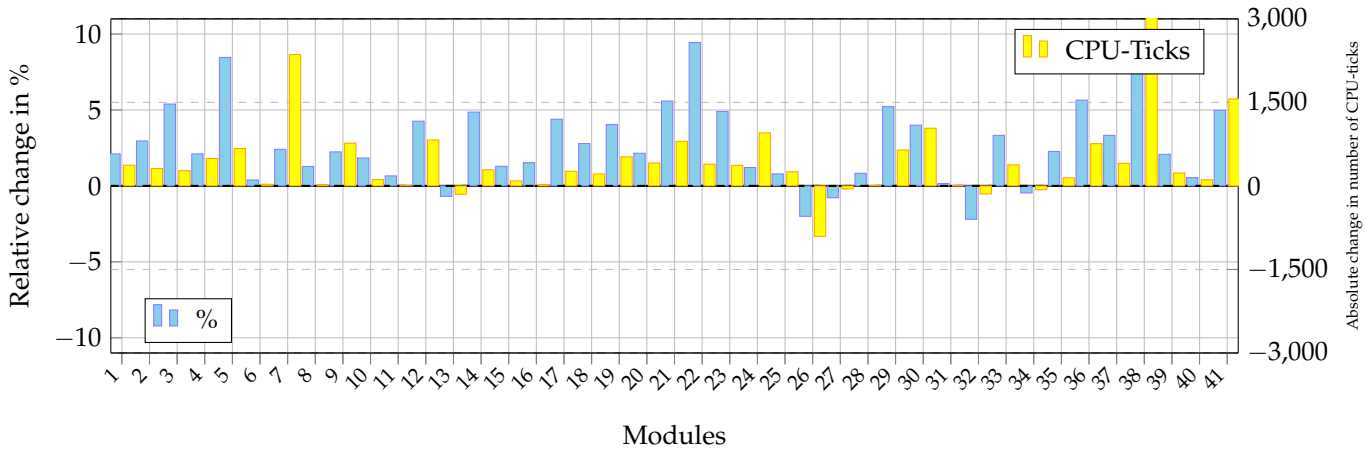


Figure 4.6: Average change in percent and CPU-ticks per module performed by block reordering on emulated hardware. *The CPU-tick column of Module 38 continues to 14921 and has been cut of in order to make the other modules column visible.*

The single test with the greatest increase of WCET can be found within *Module 38*. This test had an increase of 32143 CPU-ticks, when executed by the emulated hardware, representing an increase of 12.83%. Looking at the greatest decrease of a single tests WCET, the compiler managed to decrease the execution time with 974 CPU-ticks. This results in a change of 2.17% for *Module 26*.

The average WCET change of *Module 2* on dedicated hardware is worsened with 115.52 CPU-ticks or 1.11% as seen in 4.7. *Module 27* has its WCET decreased with 93.25 CPU-ticks which is and improvement by 1.14%.

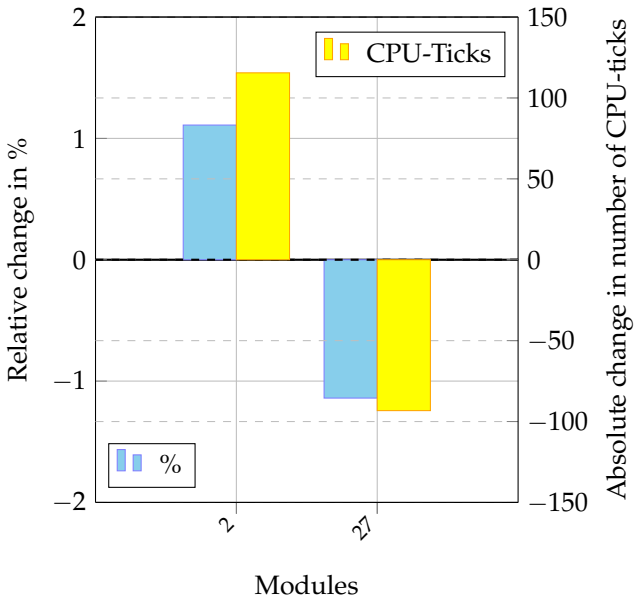


Figure 4.7: Average change in percent and CPU-ticks per module performed by block reordering on dedicated hardware.

The worst case for the dedicated hardware resulted in a smaller decrease of WCET than the worst case for the emulated hardware. The dedicated hardware resulted in 293 CPU-ticks increase which equals a increase of WCET with 0.0218%. The *Module 27* resulting in the dedicated hardware has an improvement of WCET of 1021 CPU-ticks which is equivalent to a decrease of 0.1279%.

In total a `-freorder-blocks` ended up with a increase of 272861 CPU-ticks with the emulated hardware, which is a total increase of 5.70%.

In most of the cases `-freorder-blocks` seem to reorder the basic blocks as it promised. It was also noted that the optimization often reordered the order of the instructions within these basic blocks. Other changes are in related to branching, where elimination of branching happens in a few cases. The remaining branch predictions were set to negative, meaning that the compiler thinks that the branch is unlikely to be taken.

Module 2 is in most cases identical to the *original set* with the exception of which registers are being used by instructions. The existing branch instructions had its prediction changed to positive in many cases which is in contrast to the other modules being set to negative.

Within the object code of *Module 5*, many store instructions had been replaced with register move instructions. There were also a few eliminations of load instructions.

For *Module 30*, the object code were reordered to such a degree that made it very difficult to find any similarities with the object code from the *original set*.

4.1.4 Expensive optimization

The module averages of the flag `-fexpensive-optimizations` has the largest spreading of the different test sets. The module averages for `-fexpensive-optimizations` can be found in 4.8. The largest change of CPU-tick is seen in *Module 38* with decrease of WCET of 2844.31 CPU-ticks which is equal to 1.60%. When looking at the largest change in percent, *Module 11* has a 9.62% decrease of WCET which is a change of 145 CPU-ticks.

The emulated hardware ended up having both its greatest increase and decrease of WCET in *Module 38* when looking at individual test cases. Its increase of 4380 CPU-ticks represents an worsening of 1.92% while its decrease of 9133 CPU-ticks equals an improvement of 3.70%.

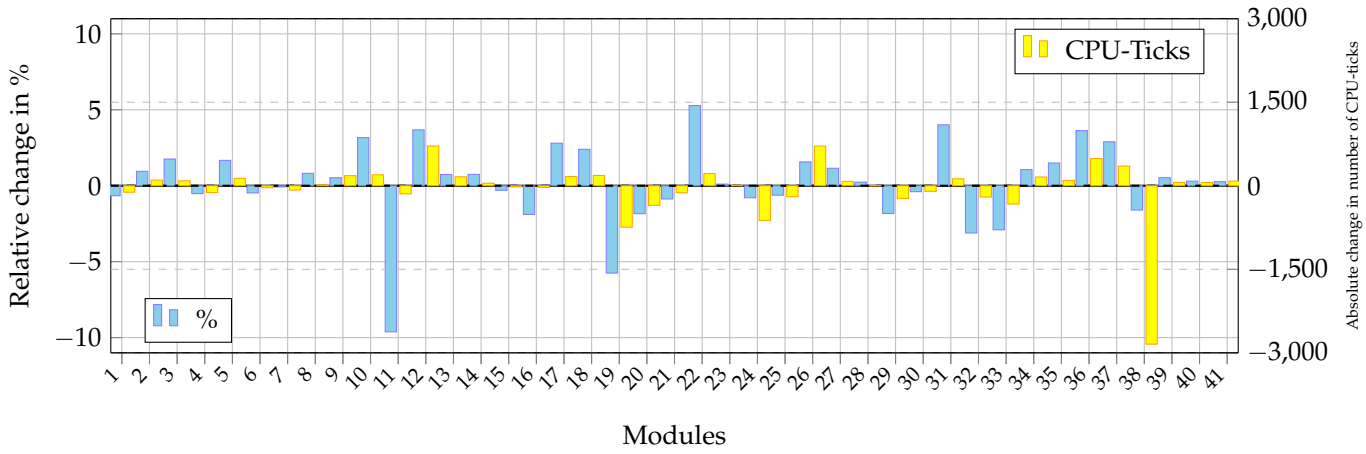


Figure 4.8: Average change in percent and CPU-ticks per module performed by `-fexpensive-optimizations` on emulated hardware.

The average change of `-fexpensive-optimizations` on dedicated hardware can be seen in 4.9. In *Module 2* an increase of WCET can be seen by 25.26 CPU-ticks which is the same as an increase of 0.24%. *Module 27* has its WCET improved by 35.17 CPU-ticks or 0.59%.

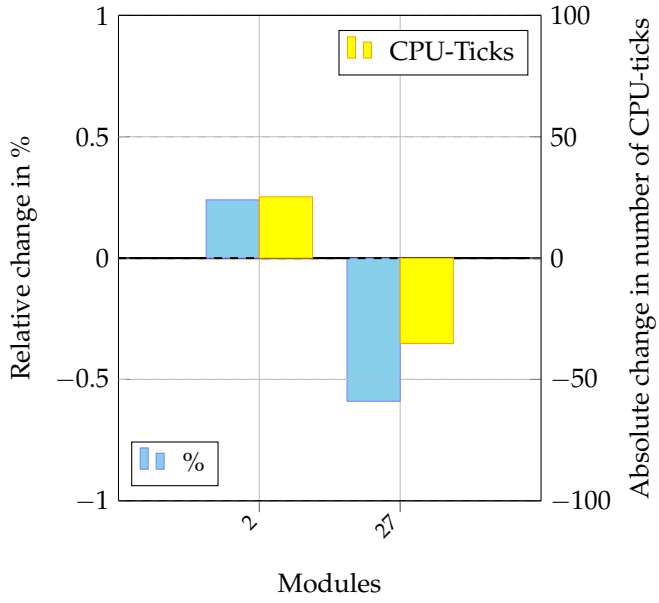


Figure 4.9: Average change in percent and CPU-ticks per module performed by `-fexpensive-optimizations` on dedicated hardware.

The worst increase of WCET by dedicated hardware can be found in *Module 2* with a value of 54 CPU-ticks which is equal to a 0.0055% increase. The best improvement of WCET is a decrease of 90 CPU-ticks. This is an decrease of 0.0074% and can be found in *Module 27*.

The resulting impact of `-fexpensive-optimizations` was negligible with a total WCET change of 44 429 CPU-ticks or 0.93% less than the *original set* on emulated hardware.

In regard to the object code, *Module 2* is largely identical to the original set. In the majority of cases, the few differences that occur are only a change of registers.

In *Module 5*, the compiler has managed to optimize a lot of the store instructions by replacing them with register move instructions. Another change is that a few load instructions

have been introduced to the code. A part of the addition instructions have been replaced by register move instructions and vice versa. In addition to this, a few register move instructions have been eliminated and almost all branch predictions has become negative.

The object code changes in *Module 30* is similar to *Module 5* where register move instructions are eliminated and the majority of branch predictions has become negative.

Another change in the object code was that the compiler added a load instruction after branching instructions. This occurred when there already existed a load instruction before a branching instructions corresponding comparison. The added load instruction was always identical to the existing load instruction.

5 Discussion

5.1 Analyzing the Results

In this thesis the absolute value, the amount of CPU-ticks, will be of a greater importance than the percentile change of WCET. The percentile change is an indicator on how big of a difference the optimization made for the module regarding to the modules size. If either the difference for the absolute or relative value is large the module and/or test can be worked upon, but in the end, the result for the amount of CPU-ticks is what matters the most. If it occurs a change for the percentage of WCET, the result could look better or worse if one were to disregard the absolute value of CPU-ticks. For example *Module 11* in 4.8 have a large increase for the relative value, but when studying the absolute value, we can see that it does not make any significant difference.

5.1.1 The certainty of our values

In this section we will discuss the certainty of our values between different test runs with the same set of flags and between the emulated and dedicated hardware.

5.1.1.1 Comparison between the dedicated and emulated hardware

As mentioned in chapter 2.2 the dedicated hardware is more accurate than the emulated hardware. To find out how "inaccurate" the emulated hardware were we compared the results of the same module from the two different hardware. In Figure 5.1 the differences between the emulated and the dedicated hardware is shown for two different modules, module 2 and 27, each with four different optimization flag bases. In Table 5.1 the values of the results and their difference in percentage units is shown. Some of the results, *align 2* and *CSE 2*, are barely 0.1 percentage points apart, while others are nearly 3 percentage points apart, *align 27* and *CSE 27*.

Some modules can differ in their behaviour, if it occurs an increase or decrease, depending on which hardware is used, for example *CSE 27*. This could be a result of differences between two test runs, read further in chapter 5.1.1.2 for a more in depth explanation. The difference in behaviour can also be a result of the different hardware used for the emulated and dedicated hardware. In the emulated hardware the only part that is identical to the dedicated hardware is the CPU. This could result in that instruction such as memory access could impact the

execution time differently due to, for example, different OS. If we consider the difference for the amount of CPU-ticks, we can see that the largest difference is for the module is in fact *CSE 27* with a value of 209.17 ticks. This is a very small amount which we in this thesis can consider insignificant.

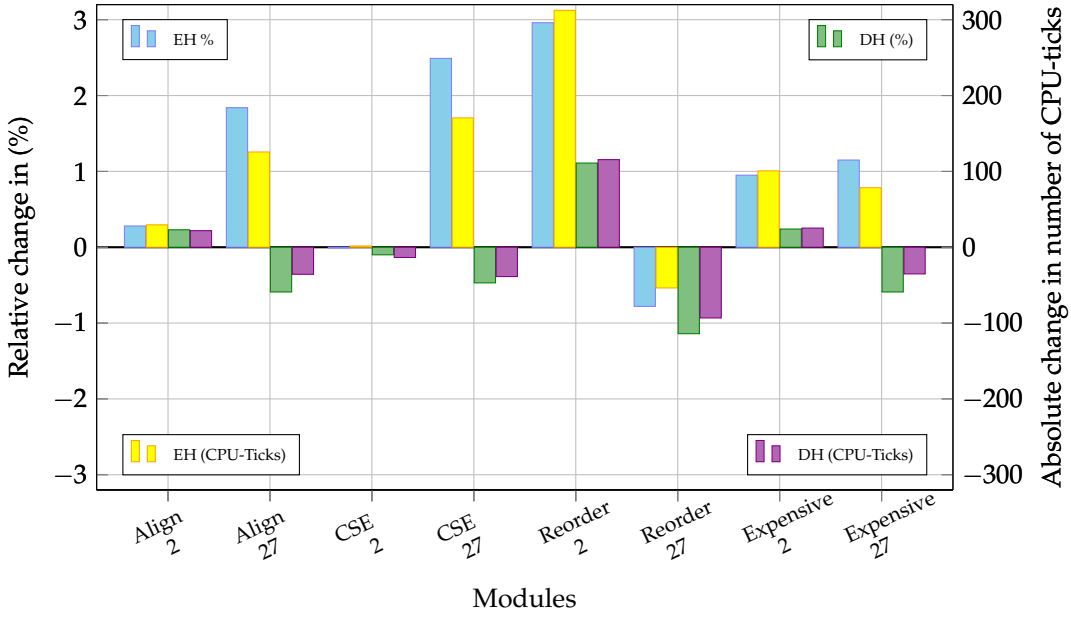


Figure 5.1: Relation between the dedicated (DH) and the emulated hardware's (EH) average change in percent and CPU-ticks.

		%			CPU-Ticks		
Test set	Module	EH	DH	|EH-DH|	EH	DH	|EH-DH|
Align	2	0.2	0.23	0.05	29.50	21.83	7.76
	27	1.84	-0.59	2.43	125.58	-35.67	161.25
CSE	2	-0.01	-0.10	0.09	1.67	-13.52	15.19
	27	2.49	-0.47	2.96	170.50	-38.67	209.17
Reorder	2	2.96	1.11	1.85	312.33	115.52	196.81
	27	-0.78	-1.14	0.36	-53.67	93.25	146.92
Expensive	2	0.95	0.24	0.71	100.67	25.26	75.41
	27	1.15	-0.59	1.74	78.42	-35.17	113.59

Table 5.1: Values and differences regarding to figure 5.1.

As stated when the dedicated hardware was presented in chapter 2.2, the dedicated hardware was very expensive to use. Due to this fact, only a few selected modules could be tested. The chosen modules 2 and 27 was chosen because they include large number of tests in relation to the other modules.

5.1.1.2 Comparison between different test runs

Ideally it should not be any WCET difference between two runs of identical code. For this thesis we compared two identical test runs, which we will go into detail in chapter 5.1.2. In figure 5.2 we can see that differences do occur between the two measured test runs. The differences can vary in size, for example from the smallest value in figure 5.2, 0.24 percent units for *test 20* to the largest difference 5.67 percent units, for *test 40*. The different test runs

can, in a few cases, change from an increased WCET to a decreased WCET, for example *test 1*. But in this case, even though the direction is changing, the actual difference $\pm \approx 100$ CPU-Ticks, is insignificant in comparison with the execution time of the larger jobs, which reach 200 000 ticks.

The results for *test 40* shows that there is a large decrease in WCET, 5.61%, for the second test run and a tiny increase in WCET, 0.06%, for the first test run. When studying figure 5.2 we can see that the values for the second test run for *test 40* stands out, compared to both the other tests and with its corresponding first test run. The change in increase/decrease of WCET is resulted by other element outside of the code for the test, for example execution of interrupt calls. One must also consider that the tests increase is 0.06% or 8 CPU-ticks which is barely considered as an increase. The only identical component for the emulated hardware compared to the dedicated hardware is the CPU, which will results in faults and/or errors in other part of the hardware. One example of other faults and/or errors is that the time difference on the emulated hardware happens outside of the control of the test code, such as context switches or memory access of different forms. Both of the two test runs values can be found in the table and figures under the subsections 7.3.1.1 and 7.3.1.2. The values of these graphs can be found in the graphs 7.1 to 7.6 with their corresponding table.

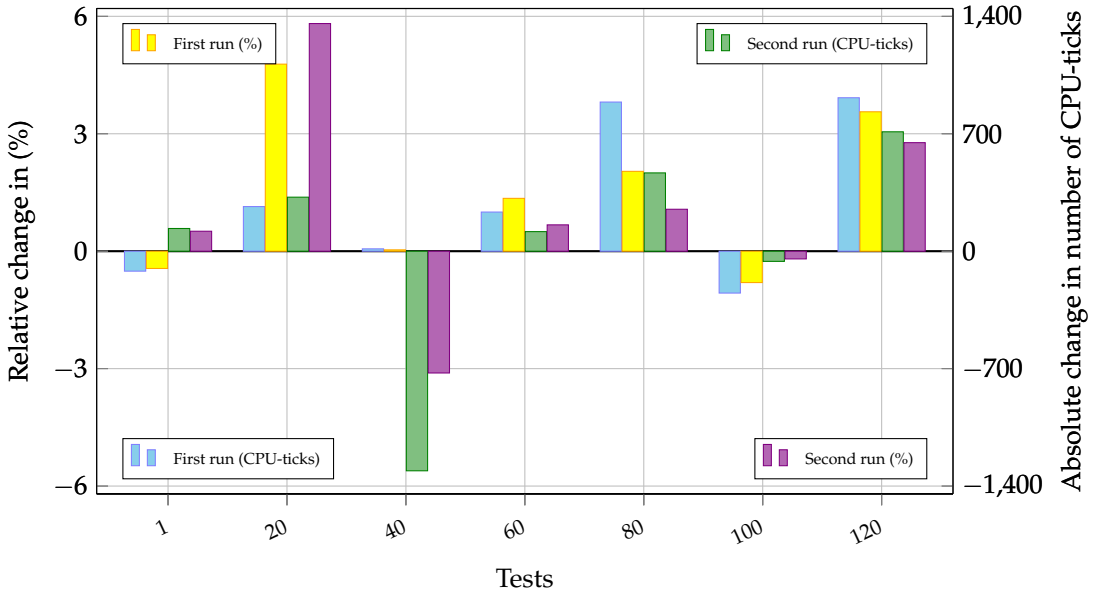


Figure 5.2: Relation between two identical testruns.

	%			CPU-Ticks		
Test	First	Second	|First-Second|	First	Second	|First-Second|
1	-0.51	0.58	1.09	-103	119	222
20	1.14	1.38	0.24	1115	1357	242
40	0.06	-5.61	5.67	8	-726	734
60	1.00	0.50	0.5	315	157	158
80	3.81	2.00	1.81	476	250	226
100	-1.07	-0.26	0.81	-187	-46	233
120	3.92	3.05	0.87	831	647	184

Table 5.2: Values and differences regarding to figure 5.2

5.1.1.3 Unmeasured impact of the GCC flags

There might be a possibility that the optimization flags used in this thesis improved the execution time of the applications in the average case, but we only care for the worst executed case out of many executions since we are looking at optimizations for a hard real-time system. Therefor we might not have seen the impact the optimisations that might have performed in the code branches where the greatest impact on execution time occurred. The branches with the highest WCET might be unaffected by the optimizations.

5.1.2 The align flags

The resulting machine code from the *align flags* is identical to the *original sets* machine code. The reason is that the machine's default for the value `n` is 4 due to the fact that the machine code uses a *32 bit instruction set*. A 32 bit instruction set will automatically align its code every fourth byte since 32 bits is equal to 4 bytes. This means that the default value chosen by the compiler and our chosen value is the same. The fact that the machine code had a 32 bit instruction set was not known at the time when the flag and its value was decided upon. The lack of time after the instruction sets size was known resulted in that further values of `n` was not tested.

Due to the fact that the *original set* and the two sets where the align flags were used resulted in an identical code, the results can be seen as if it was executed with the same set of optimization flags. These two test sets were used to compare the differences between two test runs in chapter 5.1.1.2.

The fact that the align flags are best used together are mentioned within the GCC-manual [8]. But we never found any information if the other optimization flags gain anything if they are used in a specific flag combination.

5.1.3 Common Subexpression Elimination

When studying the result from the emulated hardware in figure 4.4, one can see that there is a clear trend of increase of the WCET and that the few occurring improvements are minor. To further strengthen the conclusion that the improvements are minor, the largest decrease of WCET for the dedicated hardware for this test run is illustrated in figure 5.1 with an decrease of 38.76 CPU-ticks and 0.47%.

The fact that common subexpression elimination resulted in an almost identical machine code was unexpected because the test run resulted in a difference in WCET. This could possible be because the code standard being used prevents this type of optimization to perform to its full potential. Another reason could be that common subexpression elimination has a small impact on machine code in general.

5.1.4 Block reordering

From the result in figure 4.6, it is obvious that `-freorder-blocks` did not improve the WCET as expected. The flag managed to make the execution time slower in almost every case, especially in *Module 38*. *Module 38*'s result is an abnormality compared to the other modules, since its WCET increased with 14921 CPU-ticks which is 6.33 times higher than the second largest WCET change performed by `-freorder-blocks`. This is probably because the tests for this module are the largest that were provided. This module includes several tests that are larger than 200 000 CPU-ticks and some even reaching values of 250 000 CPU-ticks. For comparison, the largest test that is not included in this module is below 100 000 CPU-ticks. The size of the tests combined with a bad result results in the behaviour of *Module 38*.

One potential reason for the slower WCET by `-freorder-blocks` could be that the compiler managed to eliminate branching by making the program execute the instructions

in said branch every time the code was executed. This could lead to time consumed by a mispredicted branch is longer than the time to change branch.

Since `-freorder-blocks` tries to eliminate branching and changed almost every branching prediction to the negative, another reason for a worse WCET could be that these predictions are wrong. This would lead to the wrong instructions to be fetched at first and a need to fetch the correct ones, which would add extra time to the execution.

In regard of *Module 30*, it is difficult to mention any specific differences within the machine code since it has been reordered to such a degree that it's impossible for us to see any similarities. This is amplified by the reordering of instructions within the basic blocks themselves.

5.1.5 Expensive optimization

The flag with the most promising result, in regards of WCET, that has been used for this thesis is the optimization flag `-fexpensive-optimizations`. Looking at the averages in figure:4.8 the number of modules where the WCET is improved is roughly the same as for the modules with a worsened WCET. While this is not particularly good, it is the one flag we tested that had any positive impact on WCET. Unfortunately the impact is quite low, with the exception of *Module 38*.

In regards to the fact that the compiler is repeating identical load instructions after branching instructions, there is no obvious reason for the repetition to be added. When we studied the machine code we were especially looking for signs that indicated if the introduced instruction was the target of a branching instruction. In this case, that was not true and we draw the conclusion that the introduced load instruction is probably a faulty code optimization done by the compiler.

Since the second load instruction fetch the same value to the same register, this will only lead to an increased execution time. An example pseudo code of the repeating load can be seen in figure 5.3.

```
load              r0,1
compare with int  r1,r0,0
branch if equal-  r1,80
load              r0,1
```

Figure 5.3: Repeating load instruction

5.2 Method

In this section we discuss and reflect on the different parts of our method.

5.2.1 Pre-study

When studying the flags, we quickly noted that there were not much documentation about regarding the different optimization flags. This resulted in that we had to make assumptions of what the optimizations flags actually did. Since we did not have time to test all combination with the optimization flags combined with the lack of information we cannot be sure that we utilized the compilers ability fully.

5.2.2 Testing

The majority of tests were executed on the emulated hardware which resulted in a fairly accurate result with a difference in a maximum change of $\pm$ 3% 5.1. To have a more accurate result we could have executed more tests on the dedicated hardware.

For this thesis we had a few deliminations which made it hard for us to give an accurate result. As mentioned in the introduction of this particular thesis the greatest deliminations were that we executed the tests on one system, with a particularly code standard and with one type of compiler. This means that a different system can not be expected to achieve the same result as the one we have presented. This gave us an result that will work the best for this type of environment, while another user could get a different result and/or behaviour.

We did not run a test run with the tests where we were not using any optimization flags, every test run included the original set of flags used by Saab. One thing that we could have tried is running the tests without any optimization flags, to see how much improvement the original flags actually made.

5.2.3 Deficiencies in the analyzed results

When analyzing the test results we only had the change in the amount of CPU-ticks and the machine code to analyze. We did not find the time to analyze the source code to see in which cases the optimization flags were the most effective. Even though one requirement from DO-178b is that the machine code for an application must be traceable to the source code, it would greatly benefit the analysis to look at the code standard.

5.2.4 Sources

For the main part of this thesis we have a few well known sources, *the GCC-manual* [8] and *Compilers : principles, techniques & tools*, also known as *the dragon book*, [2]. *The GCC-manual* is the official manual for GCC compilers and *Compilers: principles, techniques & tools* is a book that is often used and referenced when studying compilers. A few sources has been recommended from Saabs behalf, such as the guideline DO-178b [1] and the lecture *Mutation Testing at Saab* [11]. Information, such as explanation on different computing terms et cetera, could be found within blog posts and websites. These sources are not as big or well known as some of our other sources, but they are reliable and serve their purpose. Information received from our supervisors will not be included in the bibliography, but is marked with a footnote.

5.3 Societal and ethical aspects for the thesis

The authors do not believe that the results and information in this thesis could have a direct consequence for individuals or societies safety.

As part of performing this thesis, the authors went through a background and security check, since Saab work with classified information regarding the Swedish authorities and national security. Saab is a security and defense company and sometimes sensitive information is handled that must be protected in accordance to Swedish law, offentlighets- och sekretesslagen (2009:400) [12].

6 Conclusion

6.1 The effect of our chosen optimizations

Our hypothesis for this thesis was that the optimizations would decrease the WCET for the applications we executed. Unfortunately our chosen optimization flags resulted in an increase of WCET, except for a few cases. There might be many reasons for this. One reason might be that optimizations are made for the average execution time of a program. In this thesis we only measured the WCET, resulting in that the average execution was not investigated and this hypothesis can thus not be confirmed. Another reason might be that the compiler is hindered by the fact that the source code is constructed to increase the traceability of the resulting machine code.

Our conclusion is that the GCC optimization flags used in this thesis did not have any major impact on Saab's system in regard of WCET using their compiler, neither positive or negative. Therefor we believe that Saab have managed to design a system of software where the compilers optimizations may have a minor impact of the resulting object code, through a strong design and code standard. This low optimization impact could be seen as a positive aspect of Saab's system, since it is a safety critical system that follows the DO-178B standard.

For our chosen optimization flags there were none that were of an extra interest in regard to a decrease of WCET. The only optimization flag that gave a positive result was the `-fexpensive-optimizations` with an insignificant improvement.

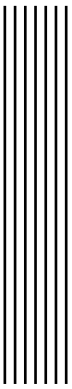
The optimization flag `-freorder-blocks` was especially bad in regard to optimization of WCET. The majority of modules had their WCET worsened.

6.2 Future work

Code developed for real-time systems often follow a specified code structure for the project and is compiled with a specific version of a compiler which may be of an older version. It could be interesting to see if a change to a more modern version of the compiler, or another type of compiler results in a improved WCET. One could also try the flags on a non real-time systems, which would not have as many strict requirements.

To have a more general result for the optimization flags, a future concept could be to execute tests of a similar structure, to measure the WCET, without the deliminations mentioned in chapter 1.4. These deliminations are that the tests were executed on one system with one

code standard with one compiler. If one were to remove just one of these deliminatons the result would probably differ.



Bibliography

[1] RTCA SC- 167 / EUROCAE WG- 12. *Software Considerations in Airborne Systems and Equipment Certification*. URL: https://www.rtca.org/.

[2] Alfred V. Aho. *Compilers : principles, techniques, & tools.* Pearson Addison-Wesley, 2007. ISBN: 0321486811. URL: https://dl.acm.org/doi/10.5555/1177220.

[3] A. Bauer and M. Pizka. *Tackling C++ Tail Calls*. URL: https://www.drdobbs.com/tackling-c-tail-calls/184401756.

[4] GCC Contributors. *15.1 Basic Blocks*. URL: https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html. (accessed: 22.05.2022).

[5] GCC Contributors. *GCC, The GNU Compiler Collection*. URL: https://gcc.gnu.org. (accessed: 22.05.2022).

[6] GCC Contributors. *Inlining of Subprograms*. URL: https://gcc.gnu.org/onlinedocs/gnat_ugn/Inlining-of-Subprograms.html. (accessed: 22.05.2022).

[7] GCC Contributors. *Inlining of Subprograms*. URL: https://gcc.gnu.org/onlinedocs/gcc-8.1.0/gcc/Common-Function-Attributes.html#Common-Function-Attributes. (accessed: 22.05.2022).

[8] GCC Contributors. *Options That Control Optimization*. URL: https://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Optimize-Options.html#Optimize-Options. (accessed: 22.05.2022).

[9] Kopetz H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Science+Business Media, 2011. ISBN: 978-1-4419-8236-0. URL: https://link.springer.com/book/10.1007/978-1-4419-8237-7.

[10] M. Jambor. "The new intraprocedural Scalar Replacement of Aggregates". In: *GCC-manual* (), pp. 1–8. URL: https://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=get&target=jambor.pdf.

[11] Pettersson N. and Brännström J. *Mutation Testing at Saab*. URL: https://www.ida.liu.se/~TDDD04/lectures/slides/2019/08-TDDD04-Mutation_testing_at_Saab.pdf. (accessed: 12.21.2021).

[12] The Swedish Parliament. *Offentlighets- och sekretesslag (2009:400)*. URL: https://www.riksdagen.se/sv/dokument-lagar/dokument/svensk-forfattningssamling/offentlighets--och-sekretesslag-2009400_sfs-2009-400. (accessed: 03.08.2022).

[13] William Von Hagen. *The definitive guide to GCC, second edition.* Apress, 2006. ISBN: 1590595858. URL: https://sensperiodit.files.wordpress.com/2011/04/hagen-the-definitive-guide-to-gcc-2e-apress-2006.pdf.

[14] M. N. Wegman and F. K. Zadeck. "Constant Propagation With Conditional Branches". In: *ACM Transactions on Programming Language and Systems* 13.2 (1991), pp. 1–30. ISSN: 0922-6443. URL: https://www.cs.utexas.edu/users/lin/cs380c/wegman.pdf.

[15] R. et al. Wilhelm. "The worst-case execution-time problem-overview of methods and survey of tools." In: *Transactions on Embedded Computing Systems* 7.3 (2008). ISSN: 15399087. URL: https://dl.acm.org/doi/10.1145/1347375.1347389.

7 Appendix

7.1 Appendix: Optimization flags

Listed below are the optimization flags used in our thesis. To be able to use the flags the hyphen before the flag must be used. For example if one wish to use the fauto-inc-dec flag it must be written as `-fauto-inc-dec` to be able to use it. Each flag has an opposite flag called fno flag. For example, if one wants to use `-O1` but turn off the fauto-inc-dec the `-fno-auto-inc-dec` flag is used in combination with `-O1`. The information of the flags has been taken from the GCC-Manual [8] and from the Definitive Guide To GCC [13]. Additional sources that are unique to that flag are referenced in the flag description. These flags are for the GCC compiler version 4.3.

7.1.1 -O1

`-O1` is a bundle of flags and contains all flags that are listed below.

`-fauto-inc-dec`

This optimization flag combines the increments and/or the decrements of the addresses with memory accesses. Not every machine supports this optimization flag.

`-fcprop-registers`

Performs a copy-propagation after the allocation of registers. Copy-propagation is when the compiler removes copies from the code. An example of copy-propagation is when:

```
a = x + y;
b = 1 + a;
```

results in:

```
b = 1 + x + y;
```

This decreases the scheduling dependencies.

`-fdce`

Dead code elimination. Removes dead code. There are different types of dead code, but the definition of a dead code is code that do not execute or execute but do not do anything useful. Example of dead code:

```
function(x, y) {
   a = x + y;
   return x;
 }
```

In this example `a = x + y` is dead code because `a` is never used for anything productive.[13]

`-fdefer-pop`

The compiler allows the arguments to stack before popping them all at once. This is not always sought after, it is more efficient to pop them one at the time, and thus the `-fno-defer-pop` flag is often used. The `-fno-defer-pop` flag pops the argument as soon as the function returns.

`-fdelayed-branch`

Attemps to reorder instructions, if the target machine supports it, to make use of the instructions slot that are available after delayed branch instructions.

`-fdse`

Dead store elimination. Removes dead stores. A dead store is when a variable is assigned a value but is not read by the following instruction. The variable occupies unnecessarily a register and is therefore a dead store. An example of how a dead store can look like:

```
a = 1;
for(i = 0 ; i < 100 ; i++) {
    [..]
}
b = 5;
a = b;
```

Here `a` is contributing to a dead store, since it could be assigned after the for-loop and since `a` is overwritten by `b`.

`-fguess-branch-probability`

The compiler guess which branch the program should take. This causes uncertainty in the object code because different runs of the application will generate different object codes. Therefore this flag is usually turned off.

`-fif-conversion`

Converts conditional jumps into branch-less versions. The flag uses the conditional moves, min, max, set flags and abs instructions.

`-fif-conversion2`

This optimization flag works the same way as `-fif-conversion` but the difference is that `-fif-conversion2` uses conditional execution to transform them into branch-less versions.

`-finline-small-functions`

When the body of a function is smaller than the call to the function, this flag inline this function. By inlining the flag replaces the call to the function with the body of the function. This flag can reduce the size of the program. [6]

`-fipa-pure-const`

Checks if a function is a pure function or a constant function. A pure function is a function where only the arguments of the function affects the return value and where the function do not have any effect except the return value [7].

`-fipa-reference`

Find static variables that can not escape the compilation unit.

`-fmerge-constants`

Merges identical variables that are constants. Instead of:

```
const a = 1;
const b = 1;
c = a + b;
```

The flag merges the constants `a` and `b`

```
const ab = 1;
c = ab + ab;
```

`-fsplit-wide-types`

If a variable occupies more than one register at a time, this flag will split those registers. An example of such variable is `long long` on a 32-bit system. Although this flag optimizes the code, it make debugging harder.

`-ftree-ccp`

Sparse conditional constant propagation on trees. This flag operates solely on local scalar variables. If one wants to operate on the memory stores and loads the flag `-ftree-store-ssp` should be used instead. Sparse conditional constant propagation removes dead code, and substituting the known value of variables, aka constant propagation. An example of constant propagation. With constant propagation:

```
a = 1;
b = a + 2;
return b;
```

will turn into:

```
return 3;
```

since `a` will always be 1, `b` will always be 3, which means the compiler can return 3. [14]

`-ftree-ch`

Loop header copying on trees. This flag saves a jump but increases the code size.

`-ftree-copyrename`

Copy renaming on trees. Rename temporary variables to names that resemble variables that are located nearby in the register.

`-ftree-dce`

Dead Code Elimination on trees. See `-fdce` for an explanation and example for dead code.

`-ftree-dominator-opts`

Perform constant propagation, copy propagation, redundancy elimination, range propagation, expression simplification and thread jumping on trees.

`-ftree-dse`

Dead store elimination on trees. See `-fdse` for an explanation and example for dead storage.

`-ftree-fre`

Full redundancy elimination on trees. FRE works in the same way as PRE, see `-ftree-pre` for an explanation and example, but the difference is that FRE only optimizes the path of expressions that are leading to the redundant instruction. This is faster than PRE, but finds less redundancies.

`-ftree-sra`

Scalar replacement of aggregates on trees. SRA is when the compiler creates new scalar variables for a part that aggregates that can not be aliased. [10]

`-ftree-ter`

Temporary expression replacement on trees. Temporary variables are replaced with their defining expression at their location.

`-funit-at-a-time`

Optimizes the code by parsing the compilation unit before producing the code. This flag has some issues, for example it may change the order of functions and variables in the code, which may introduce errors in the code.

7.1.2 -O2

`-O2` is a bundle of flags that contains the flags listed below and the flag contained in `-O1`.

`-fthread-jumps`

Check if a jump branches to an unnecessary comparison. If true, the jump is moved to another destination in the code, a point in the code that happens after the unnecessary comparison.

`-falign-functions`, `-falign-functions=`*n*

Align the functions to the next power-of-two boundary. *n* is the number of bytes the flag can skip up to. If no number is given use the machine default number. The flag `-fno-align-functions` and `-falign-functions=1` has the same effect, the functions will not be aligned.

`-falign-jumps`, `-falign-jumps=`*n*

Align the jumps to the power-of-two boundary. *n* is the number of bytes the flag can skip up to. If no number is given use the machine default number. The flag `-fno-align-jumps` and `-falign-jumps=1` has the same effect, the functions will not be aligned.

`-falign-loops`, `-falign-loops=`*n*

Align the loops to the power-of-two boundary. *n* is the number of bytes the flag can skip up to. If no number is given use the machine default number. The flag `-fno-align-loops` and `-falign-loops=1` has the same effect, the functions will not be aligned.

`-falign-labels`, `-falign-labels=`*n*

Align the branch targets to the power-of-two boundary. *n* is the number of bytes the flag can skip up to. If no number is given use the machine default number. The flag `-fno-align-labels` and `-falign-labels=1` has the same effect, the functions will not be aligned. If `-falign-loops` or `-falign-jumps` is used, the compiler will use their values instead of the ones of `-falign-labels`.

`-fcaller-saves`

The flag allows the compiler to assign values to registers that will be busy with function calls. This is only done if there is a possibility that the code will generate a better execution time.

`-fcrossjumping`

Cross-jumping transformations. This optimization flag is for decreasing the size of the code by merging equivalent code. This flag does not necessarily decrease the execution time.

`-fcse-follow-jumps`

Common sub expression elimination for jumps. Check the jump instructions to see if any jump is inaccessible. An example:

```
x = false;
if(x) {
   [..]
}
else {
   [..]
}
```

In this example the compiler will head straight to the the `else` clause because the code can not enter the `if` clause because `x` is always `false`.

`-fcse-skip-blocks`

Common sub expression elimination for blocks. Works the same as `-fcse-follow-jumps` but handles blocks instead.

`-fdelete-null-pointer-checks`

The compiler removes all null pointer checks and assume that the program will stop if the programmer decides to deference a null pointer.

`-fexpensive-optimizations`

Perform minor optimizations that are expensive in regards to processing time.

`-fgcse`

Global Common Sub expression Elimination pass, global constant and copy propagation. If the program has computed gotos the runtime may get better if one use the `-fno-gcse` flag.

`-fgcse-lm`

Is enabled when `-fgcse` is enabled. If a loop has a load instruction and a store instruction this flag moves the load to outside of the loop and keep the copy/store instruction inside the loop.

`-foptimize-sibling-calls`

Optimize sibling calls and recursive tail calls. A tail call is when the last instruction in a function is a call to another function. [2]
Example:

```
function_0(x, y) {
   [..]
   return function_1(a);
}
```

A siblingcall is when a function calls to another function with the same arguments.[3]
Example:

```
function_0(x, y) {
   [..]
   return function_1(x*2, y);
}
```

A recursive tail call is when a function recursively tail calls itself again with the arguments. [2]
Example:

```
function_0(x, y) {
   [..]
   return function_0(x*2, y);
}
```

`-fpeephole2`

Peephole optimization is when the compiler optimize a small part of the code. When that part of the code is done, the compiler starts to optimize another small part of the code.

`-fregmove`

Maximizing the amount of register tying by reassigning register numbers. This flag has the same function as the `-foptimize-register-move`

`-freorder-blocks`

Reorder blocks in the code to minimize the amount of branches taken and to enhance code locality.

`-freorder-functions`

Reorder functions in the code to enhance code locality. This flags puts the different functions into different subsections called `.text.hot` and `.text.unlikely` where `.hot` is a subsection for functions that are executed frequently and `.unlikely` where it is unlikely for a function to be executed.

`-frerun-cse-after-loop`

Rerun common sub expression elimination after loop optimization.

`-fsched-interblock`

This flag schedules instructions between basic blocks.

`-fsched-spec`

This flag use speculative motions of non-load instructions to predict the outcome. This gives an unpredictable outcome and is not recommended in high security projects.

`-fschedule-insns`

Reorder the order of the instructions to minimize stalls that occurs when an instruction is waiting for a register. Instructions that have their registers available can run instead.

`-fschedule-insns2`

Similar to `-fschedule-insns` but requires an extra pass of instruction.

`-fstrict-aliasing`

Uses the strictest form of aliasing available for the programming language. Different types of variables can share a register if they are very similar in structure. For example; an `unsigned int` can be an alias for `int` but not for a `double`.

`-fstrict-overflow`

This flag makes the compiler assume that there will not occur an overflow.

`-ftree-pre`

Partial redundancy elimination on trees. An example:

```
a = c + d;
b = c + d;
```

In the example the calculation `c + d` is redundant. To elimante the redundancy the compiler changes the code to:

```
t = c + d;
a = t;
b = t;
```

and thus removing the redundancy.

`-ftree-vrp`

Value range propagation on trees. This flag removes the need for range checks, like array checks.

7.1.3 -O3

`-O3` is a bundle of flags and contains the flags listed below and the flag contained in `-O1` and `-O2`.

`-finline-functions`

Unlike `-finline-small-functions` this flag takes simple functions into the callers and decides heuristically which functions should be integrated or not.

`-funswitch-loops`

Move invariant branches out of the loop and duplicates the loops on the branches. An example:

```
for(i = 0 ; i < 100 ; i++) {
   if(a = 5) {
      [..]
   }
   else {
      [..]
   }
}
```

will be changed into

```
if(a = 5) {
   for(i = 0 ; i < 100 ; i++) {
      [..]
   }
}
else {
   for(i = 0 ; i < 100 ; i++) {
      [..]
   }
}
```

`-fpredictive-commoning`

Predict commoning by, for example, using a loops previous computations to predict the behaviour of the loop.

`-fgcse-after-reload`

Global Common Sub expression Elimination is performed after a reload to cleanup redundancy spilling.

`-ftree-vectorize`

Loop vectorization on trees.

`-funsafe-loop-optimizations`

Assumes that the loop do not overflow and that loops with a nontrivial exit condition is infinite.

7.2 Tables that shows values for the average for the modules

Module	%	Ticks	Module	%	Ticks	Module	%	Ticks	Module	%	Ticks
1	0.22	38.67	12	4.90	950.00	23	2.01	152.00	34	0.09	14.00
2	0.10	10.33	13	0.14	31.00	24	-0.11	-84.67	35	0.24	15.33
3	-3.49	-176.50	14	5.67	338.00	25	0.76	243.19	36	3.45	463.00
4	1.02	238.00	15	-1.44	-99.00	26	0.74	334.00	37	3.63	443.00
5	4.72	374.00	16	2.01	33.00	27	2.65	181.58	38	0.20	362.88
6	-2.01	-150.50	17	1.74	104.00	28	5.21	44.00	39	2.99	332.00
7	1.38	1357.00	18	2.41	186.00	29	2.06	254.50	40	0.89	178.67
8	-2.62	-51.00	19	-5.61	-726.00	30	-0.83	-215.67	41	0.79	249.50
9	1.02	350.00	20	0.33	62.00	31	-1.38	-43.00			
10	1.27	79.00	21	-2.68	-383.00	32	0.34	22.00			
11	-8.29	-125.00	22	8.13	336.00	33	1.06	120.00			

Table 7.1: Table of average values of WCET in percentage and CPU-ticks for Align n=4

Module	%	Ticks	Module	%	Ticks	Module	%	Ticks	Module	%	Ticks
1	-0.87	-154.00	12	1.26	243.50	23	0.85	64.00	34	1.04	153.00
2	0.28	29.50	13	-0.06	-12.00	24	0.81	637.00	35	1.70	108.67
3	0.24	12.00	14	3.91	233.00	25	0.47	152.13	36	3.51	471.00
4	2.20	512.67	15	-0.68	-47.00	26	-2.00	-905.00	37	4.42	539.50
5	4.45	352.67	16	-1.83	-30.00	27	1.84	125.58	38	0.22	385.94
6	-0.70	-52.50	17	1.56	93.00	28	-2.01	-17.00	39	3.00	333.00
7	1.14	1115.00	18	1.44	111.00	29	3.62	447.50	40	1.60	320.67
8	0.41	8.00	19	0.06	8.00	30	1.79	463.00	41	2.07	649.00
9	0.58	200.00	20	0.33	62.00	31	1.41	44.00			
10	3.34	207.00	21	-2.52	-360.00	32	0.52	33.50			
11	-4.25	-64.00	22	9.78	404.00	33	2.59	294.00			

Table 7.2: Table of average values of WCET in percentage and CPU-ticks for Align default

Module	%	Ticks	Module	%	Ticks	Module	%	Ticks	Module	%	Ticks
1	1.43	252.67	12	5.31	1028.75	23	2.15	162.00	34	0.57	84.00
2	-0.01	1.67	13	-0.17	-36.00	24	2.00	1570.67	35	2.15	137.00
3	4.71	238.50	14	3.04	181.00	25	-0.16	-52.06	36	5.60	752.00
4	-1.09	-253.33	15	-0.65	-44.50	26	1.49	676.50	37	2.66	324.50
5	5.95	471.67	16	-1.10	-18.00	27	2.49	170.50	38	-0.11	-190.69
6	-1.12	-84.00	17	2.66	159.00	28	1.54	13.00	39	2.94	326.00
7	1.87	1837.00	18	2.08	160.00	29	3.38	418.00	40	0.66	133.00
8	-2.51	-49.00	19	1.53	198.00	30	1.82	471.67	41	1.06	334.00
9	0.52	178.00	20	0.62	118.00	31	3.72	116.00			
10	0.29	18.00	21	1.15	168.00	32	-1.58	-102.50			
11	-5.84	-88.00	22	5.59	231.00	33	2.16	244.50			

Table 7.3: Table of average values of WCET in percentage and CPU-ticks for Common Subexpression Elimination

Module	%	Ticks	Module	%	Ticks	Module	%	Ticks	Module	%	Ticks
1	2.11	372.33	12	4.26	825.00	23	4.90	370.00	34	-0.46	-68.00
2	2.96	312.33	13	-0.69	-150.00	24	1.21	952.33	35	2.27	145.00
3	5.39	273.05	14	4.86	289.50	25	0.79	254.38	36	5.65	758.50
4	2.11	492.33	15	1.29	89.00	26	-2.00	-907.00	37	3.33	406.50
5	8.46	671.33	16	1.53	25.00	27	-0.78	-53.67	38	8.42	14921.00
6	0.39	29.50	17	4.39	262.00	28	0.83	7.00	39	2.08	231.50
7	2.41	2359.00	18	2.79	215.00	29	5.22	645.50	40	0.55	109.67
8	1.28	25.00	19	4.04	522.00	30	4.00	1035.33	41	4.98	1562.00
9	2.24	767.00	20	2.15	410.50	31	0.16	5.00			
10	1.84	114.00	21	5.59	800.00	32	-2.20	-143.00			
11	0.66	10.00	22	9.44	390.00	33	3.33	377.50			

Table 7.4: Table of average values of WCET in percentage and CPU-ticks for Reorder Blocks

Module	%	Ticks	Module	%	Ticks	Module	%	Ticks	Module	%	Ticks
1	-0.66	-116.33	12	3.68	712.50	23	0.11	8.00	34	1.06	157.00
2	0.95	100.67	13	0.74	160.50	24	-0.79	-623.67	35	1.50	95.67
3	1.76	89.00	14	0.75	44.50	25	-0.62	-197.31	36	3.63	488.00
4	-0.52	-121.33	15	-0.30	-20.75	26	1.57	712.00	37	2.89	353.50
5	1.67	132.33	16	-1.89	-31.00	27	1.15	78.42	38	-1.60	-2844.31
6	-0.47	-35.00	17	2.80	167.00	28	0.24	2.00	39	0.53	59.25
7	-0.08	-77.00	18	2.40	185.00	29	-1.83	-226.50	40	0.30	60.33
8	0.82	16.00	19	-5.75	-743.00	30	-0.39	-101.00	41	0.27	83.50
9	0.53	181.00	20	-1.84	-351.00	31	4.01	125.00			
10	3.17	197.00	21	-0.87	-125.00	32	-3.11	-202.00			
11	-9.62	-145.00	22	5.28	218.00	33	-2.90	-329.00			

Table 7.5: Table of average values of WCET in percentage and CPU-ticks for Expensive Optimization

7.3 Graph that shows the detailed data for the test cases, WCET

7.3.1 The test runs for the Align flags sets

Table that includes the flags in these test runs.

Flags	7.3.1.1	7.3.1.2	7.3.1.3	7.3.1.4	7.3.1.5	7.3.1.6
Original	yes	yes	yes	yes	yes	yes
falign-functions	yes	yes	yes	yes	yes	yes
falign-jumps	yes	yes	yes	yes	yes	yes
falign-loops	yes	yes	yes	yes	yes	yes
falign-labels	yes	yes	yes	yes	yes	yes
fcaller-saves	no	no	no	no	no	no
fcrossjumping	no	no	no	no	no	no
fcsefollow-jumps	no	no	yes	yes	no	no
fcse-skip-blocks	no	no	yes	yes	no	no
fdelete-null-pointer-checks	no	no	no	no	no	no
fexpensive-optimizations	no	no	no	yes	yes	no
fgcse	no	no	no	no	no	no
fgcse-lm	no	no	no	no	no	no
foptimize-sibling-calls	yes	yes	yes	yes	yes	no

Table 7.6: List of flags used in the aligns flags set

7.3.1.1 Align flags with n=default

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	-0.51	-103	32	3.56	238	63	0.69	227	93	2.00	123
2	-0.85	-136	33	-0.73	-51	64	-0.65	-204	94	3.01	403
3	-1.35	-223	34	-0.66	-46	65	-2.28	-1021	95	4.00	539
4	0.28	28	35	-0.69	-48	66	-1.73	-789	96	3.78	449
5	0.21	26	36	-0.65	-43	67	4.67	526	97	5.01	630
6	0.63	60	37	-1.83	-30	68	3.98	486	98	-0.60	-106
7	0.03	3	38	1.56	93	69	3.38	415	99	0.87	144
8	0.21	26	39	1.44	111	70	-0.39	-19	100	-1.07	-187
9	0.35	34	40	0.06	8	71	-0.23	-11	101	0.52	1266
10	0.30	15	41	0.18	37	72	2.28	108	102	-0.22	-537
11	0.18	9	42	0.51	87	73	-9.57	-301	103	0.15	330
12	5.83	1010	43	-2.52	-360	74	-9.94	-312	104	-0.38	-938
13	0.86	280	44	9.78	404	75	-7.23	-219	105	-0.45	-1122
14	1.22	248	45	0.85	64	76	3.18	262	106	0.35	864
15	4.58	363	46	0.96	746	77	4.53	282	107	0.66	1376
16	4.60	366	47	0.69	531	78	3.50	290	108	2.86	423
17	4.16	329	48	0.78	634	79	-2.01	-17	109	0.61	837
18	-0.90	-67	49	-3.52	-1096	80	3.81	476	110	0.50	1235
19	-0.50	-38	50	-0.14	-47	81	3.41	419	111	0.47	1169
20	1.14	1115	51	0.13	49	82	2.36	470	112	0.10	232
21	0.41	8	52	0.46	161	83	1.49	431	113	0.47	1189
22	0.58	200	53	1.29	384	84	1.69	488	114	3.27	362
23	3.34	207	54	0.56	179	85	1.41	44	115	3.05	333
24	-4.25	-64	55	0.45	162	86	0.58	38	116	2.65	273
25	-1.35	-236	56	0.83	284	87	0.45	29	117	3.00	364
26	1.04	182	57	0.96	268	88	3.44	387	118	-0.11	-21
27	3.64	781	58	-0.01	-3	89	1.76	201	119	0.76	152
28	1.17	247	59	0.05	16	90	1.04	153	120	3.92	831
29	0.68	152	60	1.00	315	91	1.61	117	121	1.98	620
30	-0.83	-176	61	5.58	1538	92	1.49	86	122	2.16	678
31	4.35	228	62	0.68	201						

Table 7.7: Table of detailed values of WCET in percentage for Align flags with n=default

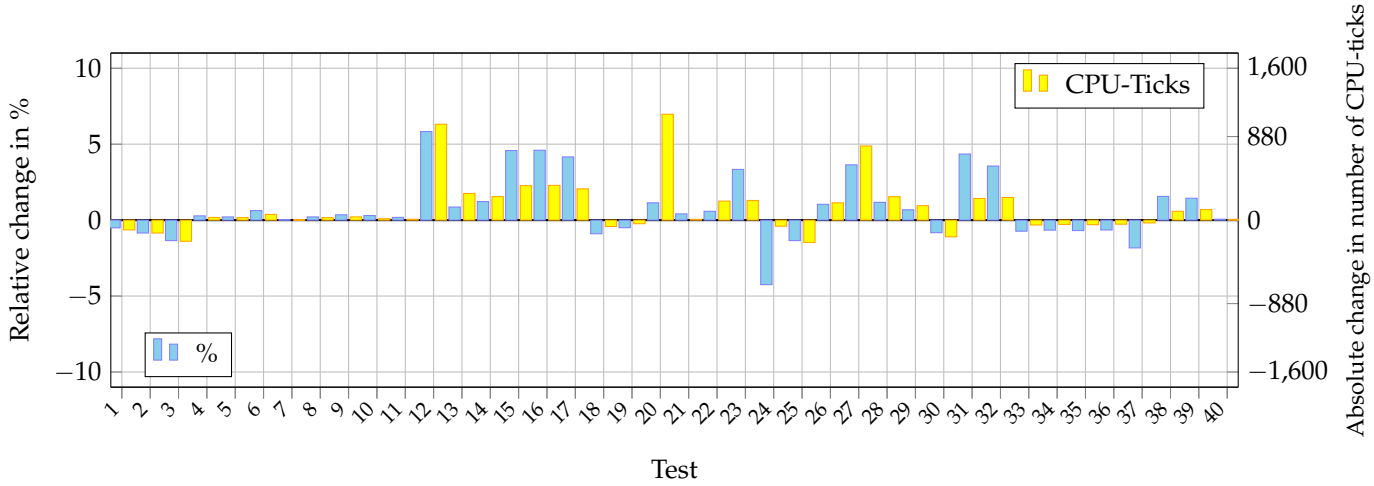


Figure 7.1: Detailed result for Align flags with n=default optimization, test 1-40

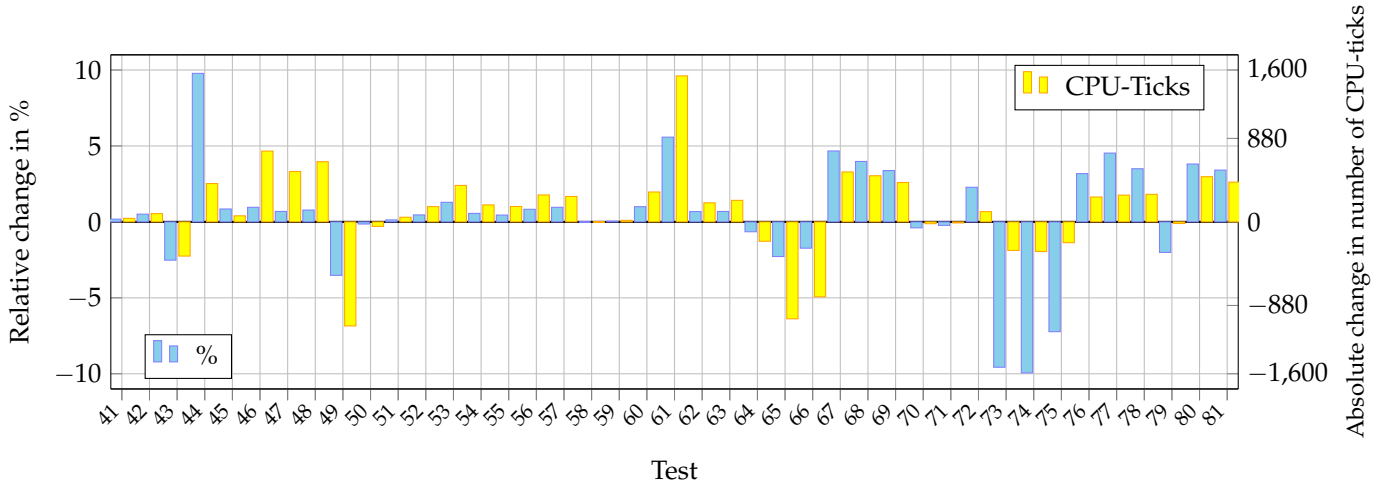


Figure 7.2: Detailed result for Align flags with n=default optimization, test 41-81

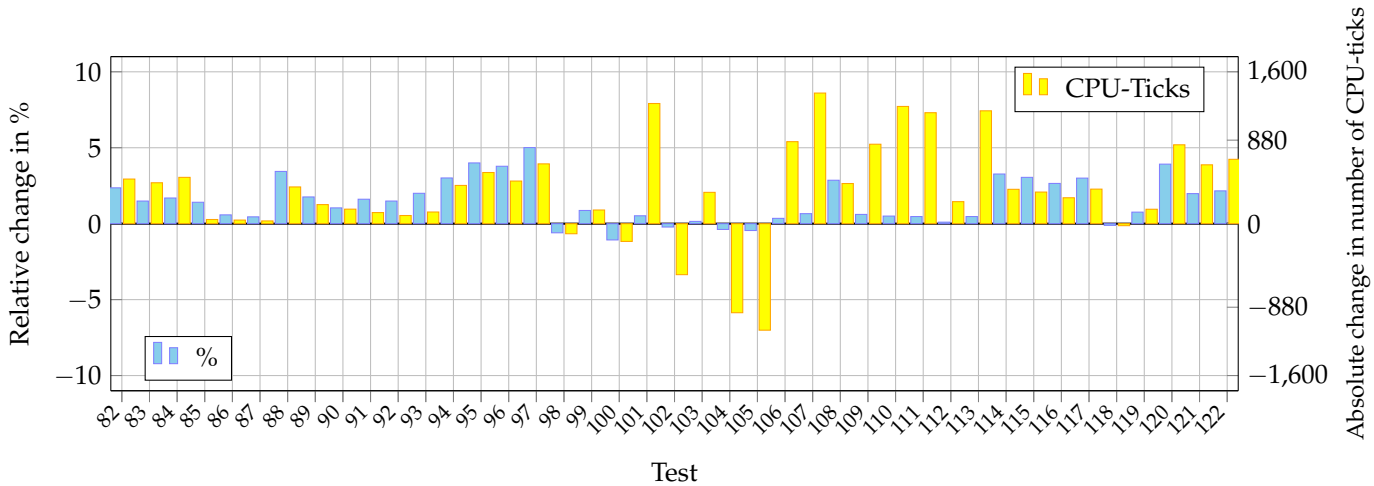


Figure 7.3: Detailed result for Align flags with n=default optimization, test 82-122

7.3.1.2 Align flags with n=4

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	0.58	119	32	4.80	321	63	0.38	126	93	0.76	47
2	-0.08	-13	33	-1.60	-112	64	0.12	38	94	3.64	487
3	0.06	10	34	-1.61	-113	65	0.27	123	95	3.26	439
4	-0.91	-90	35	-1.24	-86	66	1.19	545	96	3.02	358
5	-0.02	-2	36	-1.28	-85	67	4.66	524	97	4.20	528
6	0.68	65	37	2.01	33	68	4.70	575	98	-0.63	-112
7	0.44	44	38	1.74	104	69	3.99	490	99	1.21	201
8	-0.11	-13	39	2.41	186	70	0.10	5	100	-0.26	-46
9	0.60	58	40	-5.61	-726	71	0.40	19	101	-0.70	-1724
10	-4.27	-217	41	0.26	55	72	0.11	5	102	-0.47	-1128
11	-2.69	-136	42	0.40	69	73	-3.28	-103	103	0.10	222
12	0.82	142	43	-2.68	-383	74	-2.10	-66	104	1.17	2887
13	1.22	397	44	8.13	336	75	-0.07	-2	105	-0.62	-1538
14	0.86	175	45	2.01	152	76	3.42	282	106	0.52	1284
15	3.22	255	46	0.48	370	77	2.81	175	107	0.41	849
16	5.26	418	47	0.02	17	78	3.32	275	108	3.38	500
17	5.67	449	48	-0.79	-641	19	5.21	44	109	0.04	59
18	-1.69	-126	49	1.50	467	80	2.00	250	110	0.38	950
19	-2.32	-175	50	1.50	489	81	2.11	259	111	0.09	221
20	1.38	1357	51	0.24	88	82	-1.48	-294	112	1.21	2757
21	-2.62	-51	52	0.24	82	83	-0.63	-182	113	0.17	424
22	1.02	350	53	0.68	202	84	-0.59	-171	114	2.86	316
23	1.27	79	54	0.42	134	85	-1.38	-43	115	3.17	346
24	-8.29	-125	55	0.16	59	86	0.17	11	116	2.29	236
25	7.88	1374	56	0.29	100	87	0.51	33	117	3.54	430
26	6.19	1086	57	3.04	847	88	1.64	184	118	1.53	289
27	2.84	608	58	0.03	9	89	0.49	56	119	-1.99	-400
28	3.47	732	59	-0.16	-56	90	0.09	14	120	3.05	647
29	1.31	295	60	0.50	157	91	-1.09	-79	121	1.16	363
30	-1.10	-233	61	3.80	1046	92	1.35	78	122	0.43	136
31	6.78	355	62	0.35	103						

Table 7.8: Table of detailed values of WCET in percentage for Align flags with n=4

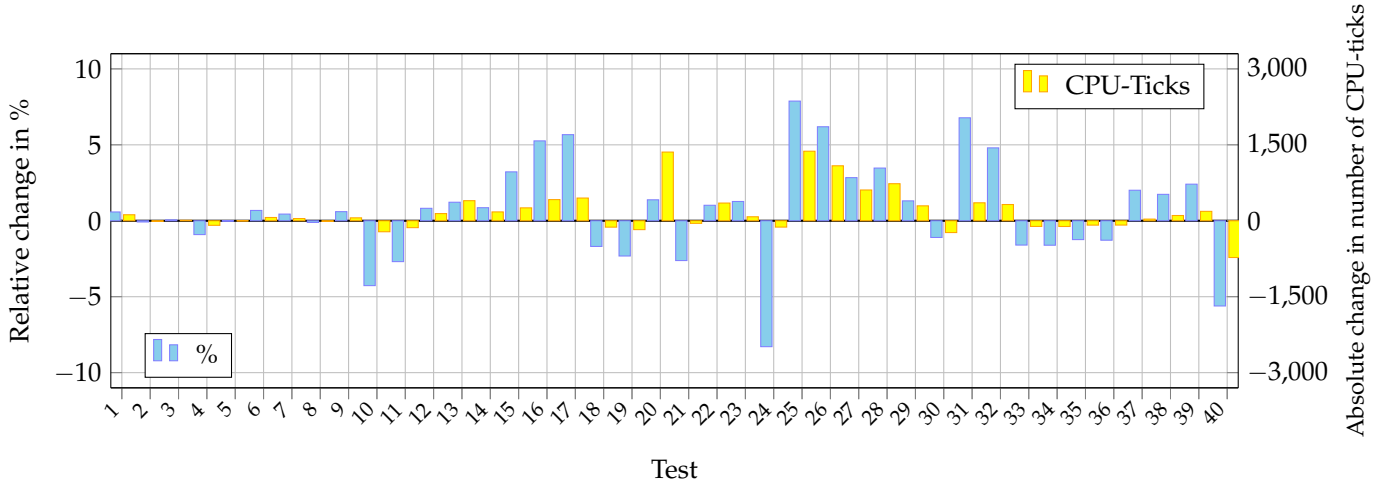


Figure 7.4: Detailed result for Align flags with n=4 optimization, test 1-40

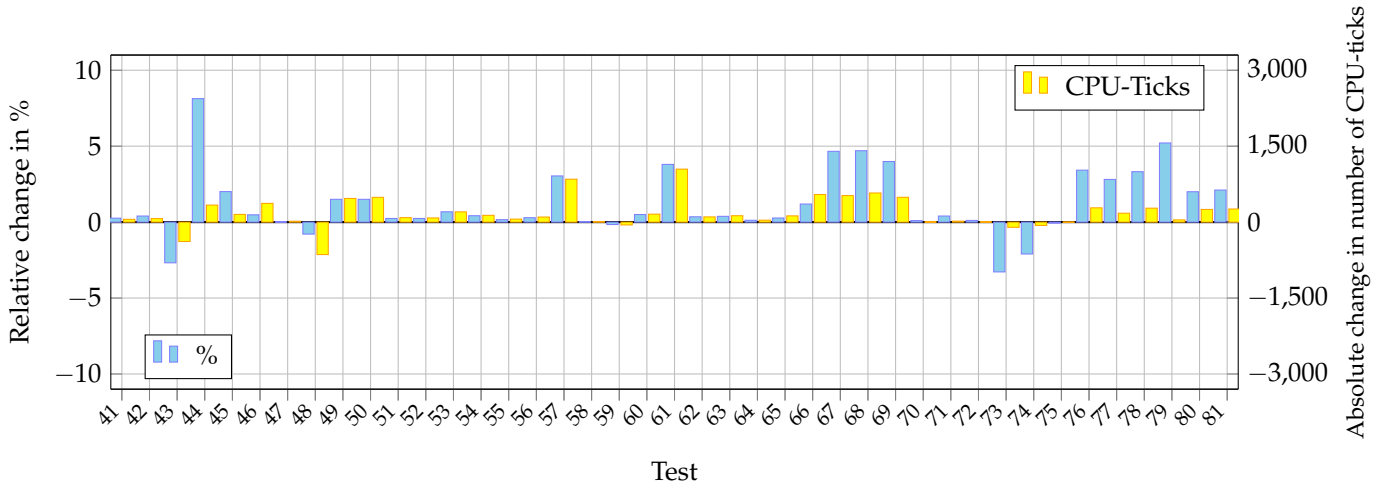


Figure 7.5: Detailed result for Align flags with n=4 optimization, test 41-81

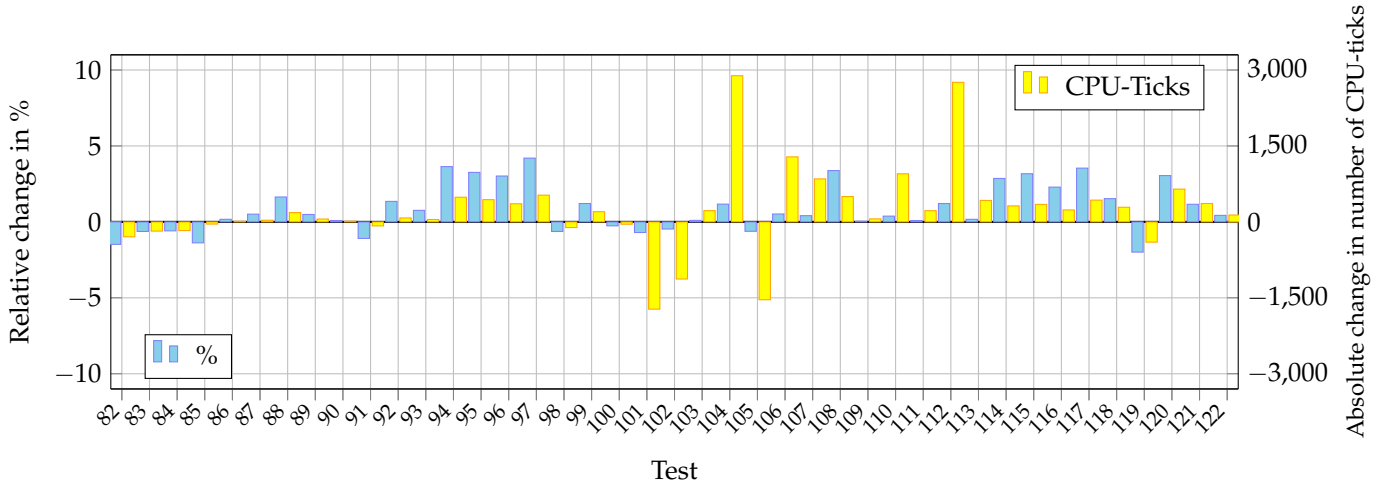


Figure 7.6: Detailed result for Align flags with n=4 optimization, test 82-122

7.3.1.3 Third Set Align flags

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	-0.01	-3	32	2.24	150	63	-0.32	-104	93	2.96	182
2	-0.95	-153	33	-1.12	-78	64	-0.79	-245	94	2.41	322
3	-1.00	-165	34	-1.17	-82	65	-0.52	-234	95	2.08	280
4	0.59	59	35	-1.08	-75	66	-0.69	-314	96	2.38	282
5	0.07	9	36	-0.89	-59	67	4.16	468	97	2.76	347
6	0.71	68	37	-0.18	-3	68	3.28	401	98	-0.91	-162
7	0.18	18	38	1.93	115	69	2.52	309	99	-1.08	-180
8	0.21	26	39	0.90	69	70	0.50	24	100	1.64	288
9	0.49	47	40	-2.57	-332	71	2.06	97	101	1.32	3247
10	-4.23	-215	41	-0.25	-52	72	2.49	118	102	1.62	3883
11	-4.02	-203	42	-4.51	-772	73	-6.01	-189	103	1.43	3138
12	4.87	843	43	-0.64	-91	74	-6.25	-196	104	1.33	3281
13	-1.32	-427	44	3.46	143	75	0.73	22	105	0.68	1703
14	0.16	33	45	-0.66	-50	76	3.64	300	106	1.59	3955
15	8.72	691	46	0.46	356	77	3.13	195	107	0.24	508
16	7.72	614	47	0.33	255	78	2.98	247	108	-0.80	-118
17	7.88	624	48	-0.94	-759	79	-1.18	-10	109	-0.29	-402
18	-0.59	-44	49	-2.68	-835	80	1.19	238	110	2.42	6032
19	-2.36	-178	50	0.25	83	81	1.03	126	111	-1.21	-2990
20	-0.05	-49	51	-0.11	-40	82	2.13	423	112	0.25	579
21	-4.87	-95	52	0.27	94	83	1.52	441	113	1.14	2864
22	-0.34	-117	53	-3.29	-983	84	1.74	502	114	2.76	305
23	1.16	72	54	-0.81	-261	85	-2.76	-86	115	1.40	15
24	-5.44	-82	55	-0.45	-161	86	1.26	82	116	2.89	298
25	5.39	940	56	-0.32	-109	87	1.22	79	117	2.71	329
26	3.33	584	57	-0.59	-165	88	3.19	359	118	-2.11	-399
27	5.29	1135	58	-0.41	-122	89	2.33	267	119	0.64	128
28	4.27	901	59	-0.92	-312	90	-0.27	-40	120	1.06	225
29	6.23	1401	60	-0.65	-206	91	1.26	91	121	0.48	150
30	-4.47	-943	61	0.37	103	92	2.38	137	122	0.73	229
31	4.3	227	62	0.36	106						

Table 7.9: Table of detailed values of WCET in percentage for Third Set Align flags

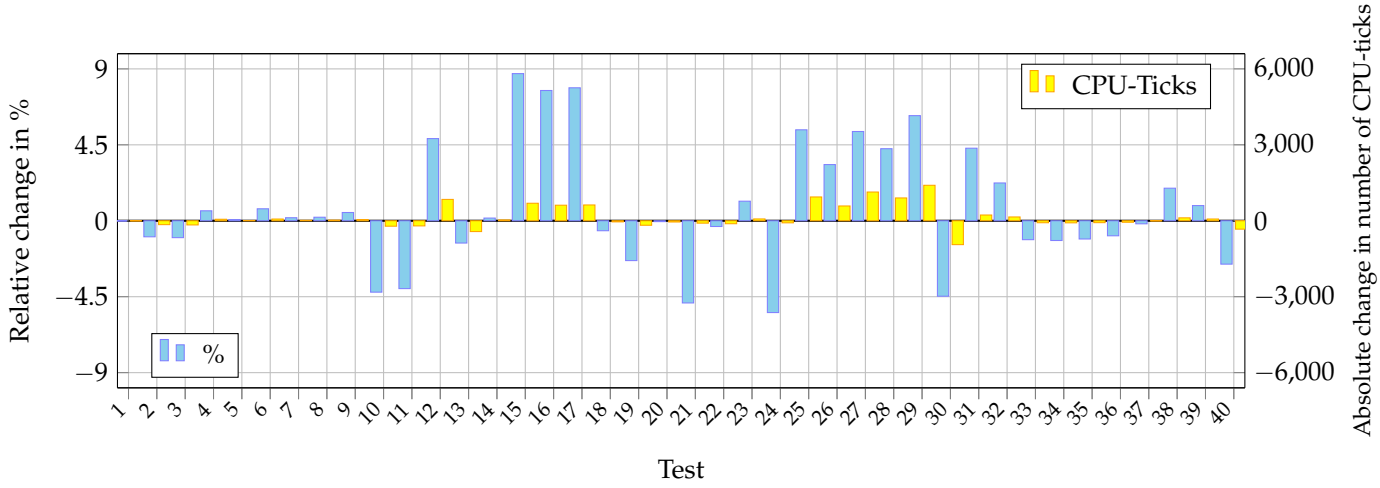


Figure 7.7: Detailed result for Third Set Align flags optimization, test 1-40

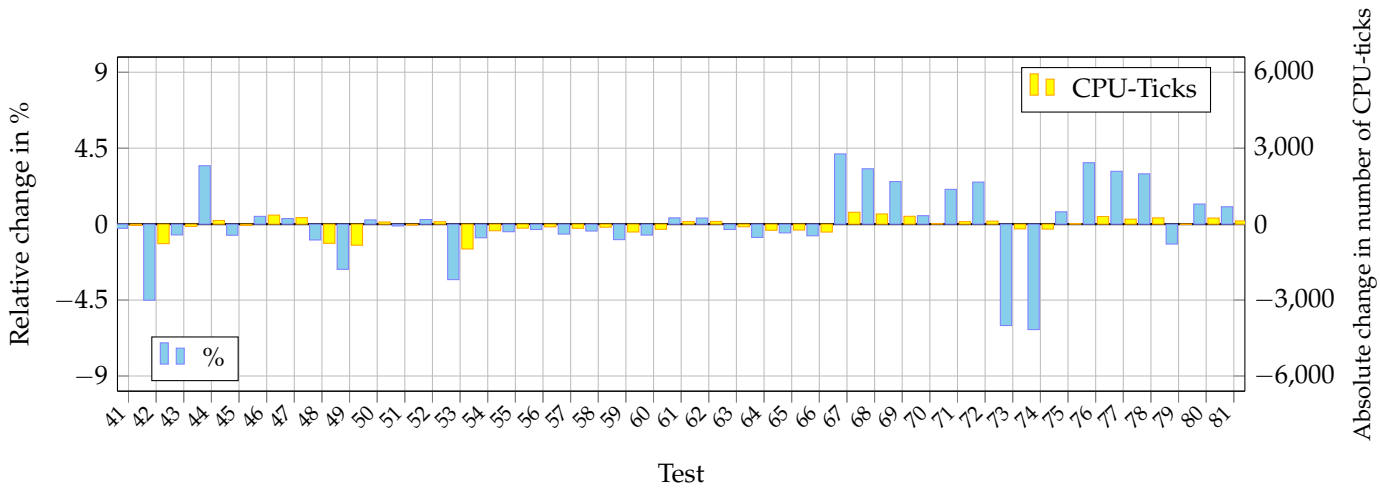


Figure 7.8: Detailed result for Third Set Align flags optimization, test 41-81

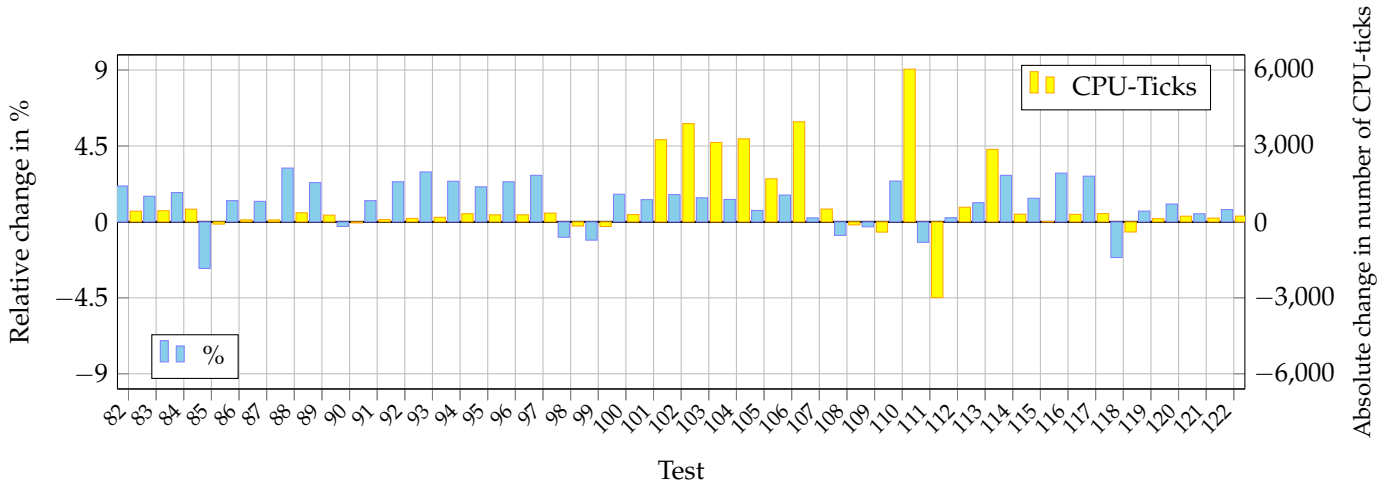


Figure 7.9: Detailed result for Third Set Align flags optimization, test 82-122

7.3.1.4 Fourth 4 Align flags

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	2.38	486	32	1.56	104	63	-1.39	-458	93	0.44	27
2	1.21	194	33	-0.73	-51	64	-1.66	-518	94	4.53	607
3	0.78	129	34	-0.54	-38	65	0.33	149	95	3.73	503
4	0.09	9	35	-0.16	-11	66	1.62	741	96	1.43	170
5	-0.70	-85	36	-0.33	-22	67	4.42	497	97	3.44	432
6	-0.13	-12	37	-2.62	-43	68	3.62	442	98	-1.16	-206
7	-0.18	-18	38	-0.20	-12	69	3.15	387	99	-2.17	-360
8	-0.71	-87	39	1.56	120	70	-8.87	-428	100	-2.22	-389
9	-0.39	-37	40	-1.79	-231	71	-8.61	-405	101	1.04	2541
10	1.16	59	41	0.08	17	72	-7.60	-361	102	1.68	4027
11	-0.93	-47	42	-4.58	-784	73	-6.27	-197	103	1.44	3172
12	0.68	118	43	-0.93	-133	74	-6.31	-198	104	2.34	5759
13	-0.97	-314	44	-3.29	-136	75	-1.62	-49	105	0.64	1601
14	0.52	105	45	-1.26	-95	76	1.49	123	106	2.31	5741
15	5.87	465	46	0.67	520	77	0.56	35	107	0.66	1373
16	6.64	528	47	1.75	1353	78	1.22	101	108	-2.01	-297
17	5.87	465	48	-0.33	-268	79	3.08	26	109	-0.37	-509
18	-1.96	-146	49	-2.13	-664	80	2.20	274	110	1.41	3524
19	-2.16	-163	50	0.85	276	81	1.86	228	111	2.49	6138
20	0.66	646	51	0.05	19	82	3.28	653	112	6.97	15902
21	2.87	56	52	0.51	178	83	1.85	538	113	1.26	3162
22	0.08	28	53	-1.91	-572	84	2.52	726	114	-0.59	-65
23	1.93	120	54	-0.16	-51	85	-3.27	-102	115	-0.32	-35
24	-2.26	-34	55	-0.35	-126	86	3.27	-88	116	-0.70	-72
25	4.26	743	56	0.18	60	87	-1.19	-77	117	-0.54	-66
26	2.29	402	57	-0.15	-43	88	1.32	149	118	1.95	368
27	5.68	1218	58	-0.38	-113	89	-0.46	-53	119	0.71	142
28	4.88	1029	59	-0.63	-214	90	1.00	147	120	1.25	265
29	-1.64	-368	60	-0.24	-75	91	-0.83	-60	121	0.51	161
30	-3.57	-753	61	-0.12	-32	92	-0.05	-3	122	0.61	193
31	2.52	132	62	0.56	465						

Table 7.10: Table of detailed values of WCET in percentage for Fourth Set Align flags

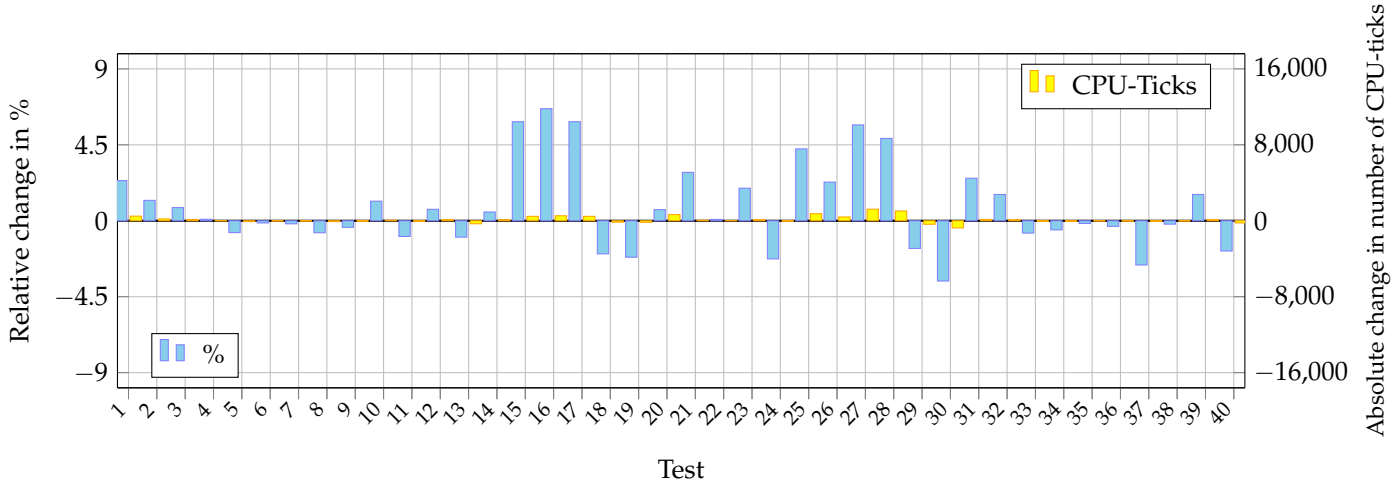


Figure 7.10: Detailed result for Fourth Set Align flags optimization, test 1-40

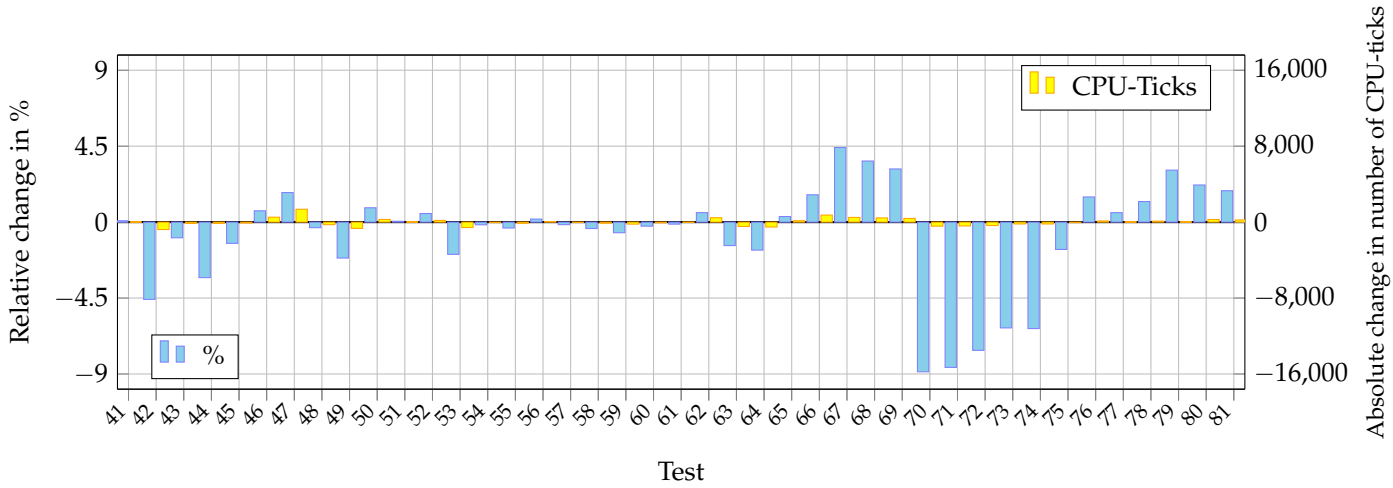


Figure 7.11: Detailed result for Fourth Set Align flags optimization, test 41-81

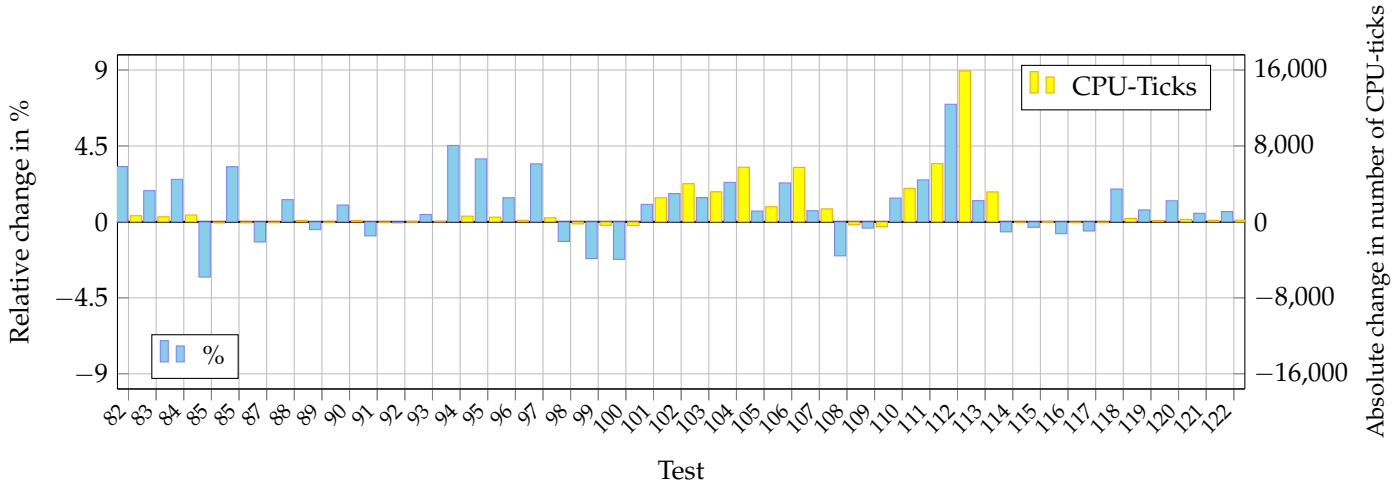


Figure 7.12: Detailed result for Fourth Set Align flags optimization, test 82-122

7.3.1.5 Align Expensive Optimization

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	2.50	510	32	-1.90	-127	63	-0.70	-232	93	-2.03	-125
2	1.69	271	33	-0.73	-51	64	-0.69	-216	94	5.06	677
3	1.29	213	34	-0.85	-60	65	1.94	870	95	6.41	864
4	-2.46	-244	35	-0.83	-58	66	1.80	823	96	2.71	322
5	-1.19	-145	36	-0.80	-53	67	7.27	818	97	3.11	391
6	-1.54	-147	37	-6.16	-101	68	5.39	659	98	-0.20	-36
7	-1.66	-165	38	3.28	196	69	5.24	643	99	1.01	167
8	-1.68	-204	39	0.47	36	70	0.81	39	100	-0.41	-72
9	-1.77	-170	40	-0.19	-24	71	1.15	54	101	-2.75	-6736
10	-1.71	-87	41	-0.70	-147	72	3.98	189	102	-1.72	-4126
11	-1.94	-98	42	-1.25	-214	73	-3.53	-111	103	-2.14	-4718
12	4.99	864	43	3.57	511	74	-2.90	-91	104	-2.06	-5072
13	-0.99	-320	44	4.77	197	75	-0.63	-19	105	-2.15	-5385
14	1.36	275	45	3.27	247	76	5.80	478	106	-1.72	-4289
15	1.50	119	46	1.74	1352	77	7.04	438	107	-0.83	-1728
16	1.62	129	47	1.50	1159	78	5.96	494	108	2.03	300
17	1.44	114	48	0.51	412	79	0.71	6	109	0.39	538
18	2.41	180	49	-1.41	-440	80	-1.75	-218	110	-1.41	-3512
19	1.51	114	50	0.93	303	81	-2.45	301	111	-3.25	-8003
20	-0.44	-429	51	-0.09	-34	82	2.80	556	112	2.66	6074
21	-1.80	-35	52	0.15	53	83	2.14	622	113	-2.45	-6139
22	1.60	549	53	-1.05	-315	84	2.00	577	114	2.53	280
23	0.85	53	54	-0.27	-88	85	1.61	50	115	1.80	197
24	-5.91	-89	55	-0.97	-352	86	-0.68	-44	116	0.88	9
25	-0.55	-96	56	-0.79	-271	87	0.02	1	117	1.66	201
26	4.91	861	57	-0.39	-108	88	1.91	215	118	2.96	558
27	4.87	1045	58	-0.28	-85	89	0.04	5	119	0.89	178
28	4.23	893	59	-0.20	-68	90	1.99	293	120	3.07	650
29	-1.00	-225	60	0.43	137	91	-4.00	-290	121	1.34	419
30	2.77	584	61	0.53	146	92	-3.05	-176	122	1.03	323
31	-0.34	-18	62	0.61	181						

Table 7.11: Table of detailed values of WCET in percentage for Align Expensive Optimization

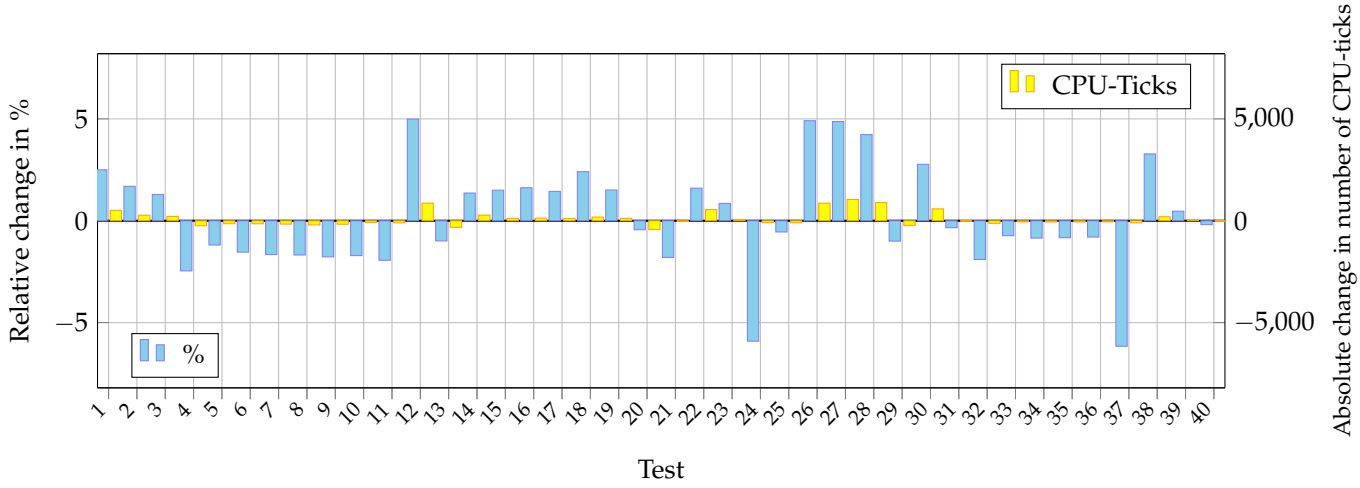


Figure 7.13: Detailed result for Align Expensive Optimization, test 1-40

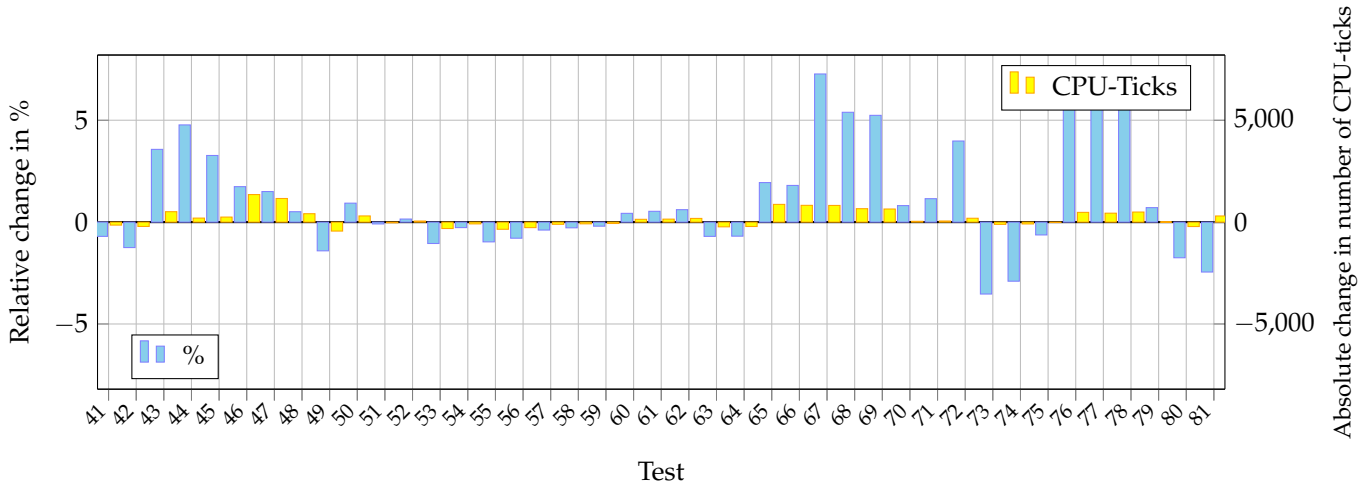


Figure 7.14: Detailed result for Align Expensive Optimization, test 41-81

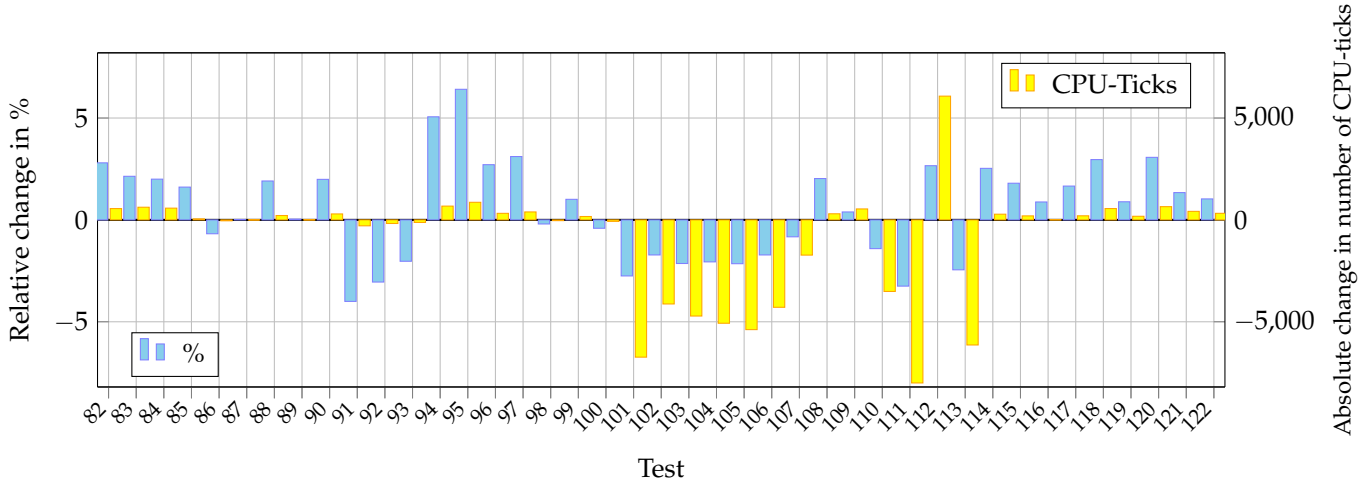


Figure 7.15: Detailed result for Align Expensive Optimization, test 82-122

7.3.1.6 Merged Align flags

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	-0.81	-166	32	1.39	93	63	0.90	298	93	1.85	114
2	-1.09	-174	33	-0.09	-6	64	0.34	107	94	4.89	655
3	-0.43	-71	34	0.14	10	65	1.65	739	95	4.90	660
4	0.73	73	35	0.50	35	66	2.16	989	96	3.24	384
5	0.70	85	36	0.15	10	67	4.92	554	97	3.47	436
6	0.58	55	37	1.34	22	68	4.14	506	98	1.86	331
7	0.21	21	38	3.18	190	69	4.12	506	99	2.91	483
8	0.62	76	39	0.70	54	70	0.91	44	100	1.28	225
9	0.32	31	40	-2.30	-297	71	0.34	16	101	-0.31	-766
10	1.40	71	41	0.39	82	72	4.15	197	102	-0.72	-1717
11	0.32	16	42	-0.46	-78	73	0.80	25	103	-0.19	-419
12	4.72	740	43	0.61	87	74	0.22	7	104	0.53	1316
13	0.09	28	44	3.73	154	75	3.76	114	105	-0.56	-1407
14	0.08	17	45	1.88	142	76	4.53	373	106	0.02	47
15	7.10	563	46	0.67	521	77	4.55	283	107	0.15	308
16	8.08	642	47	1.31	1013	78	4.29	356	108	0.05	7
17	7.35	582	48	0.67	540	79	-4.50	-38	109	0.25	341
18	-2.47	-184	49	-0.83	-257	80	3.46	432	110	1.11	2756
19	-2.10	-158	50	0.51	167	81	4.49	551	111	0.27	675
20	1.17	1149	51	-0.39	-142	82	1.36	271	112	0.80	1824
21	4.10	80	52	-0.30	-104	83	1.04	302	113	-0.10	-249
22	0.69	237	53	0.38	115	84	1.17	337	114	4.29	474
23	1.98	123	54	0.56	180	85	0.80	25	115	4.76	520
24	-6.44	-97	55	0.01	2	86	1.46	95	116	3.48	359
25	0.98	171	56	0.41	139	87	1.14	74	117	4.34	526
26	1.13	199	57	3.59	1001	88	-0.68	-77	118	1.42	268
27	2.02	434	58	0.48	143	89	-1.40	-160	119	0.93	186
28	3.68	775	59	0.25	84	90	0.26	39	120	0.72	153
29	10.88	2446	60	1.12	353	91	-0.12	-9	121	0.03	9
30	0.63	133	61	3.13	863	92	0.66	38	122	-0.18	-56
31	2.52	132	62	0.78	232						

Table 7.12: Table of detailed values of WCET in percentage for Merged Align flags

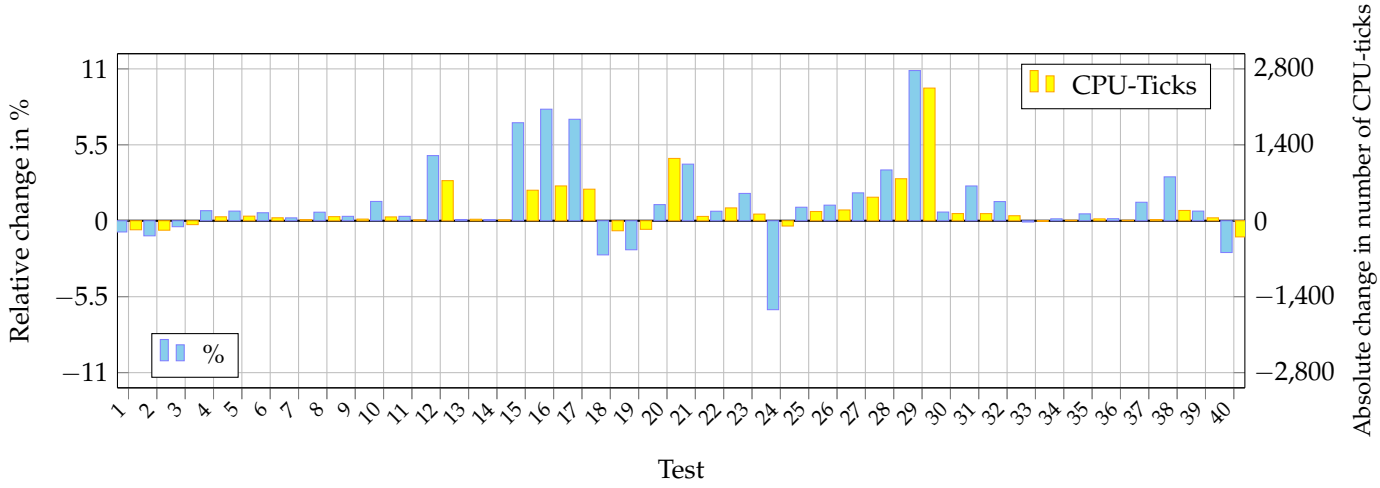


Figure 7.16: Detailed result for Merged Align flags optimization, test 1-40

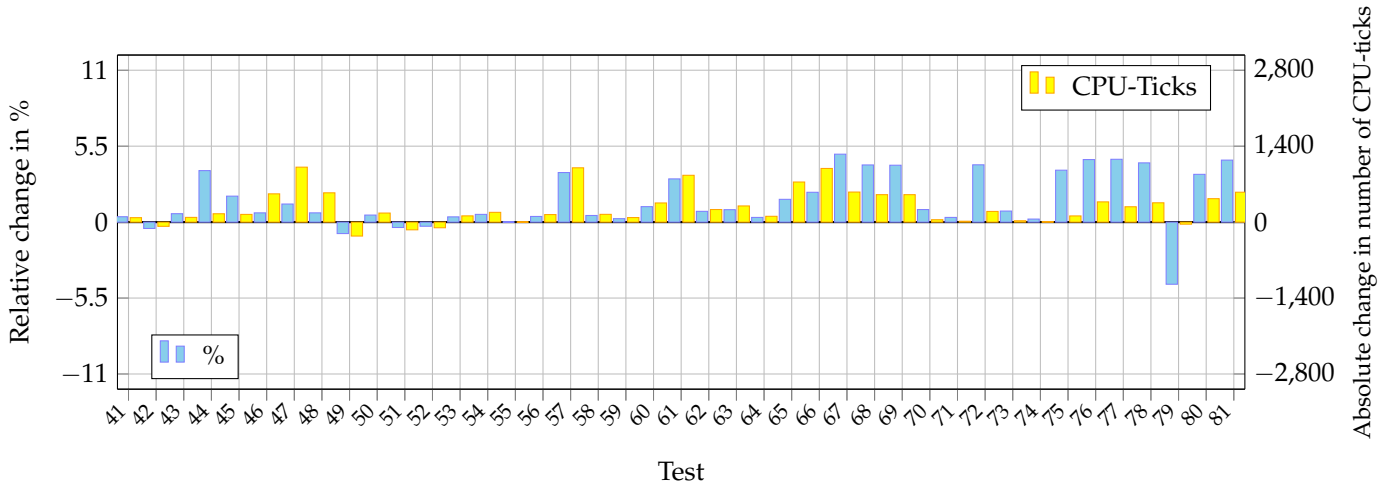


Figure 7.17: Detailed result for Merged Align flags optimization, test 41-81

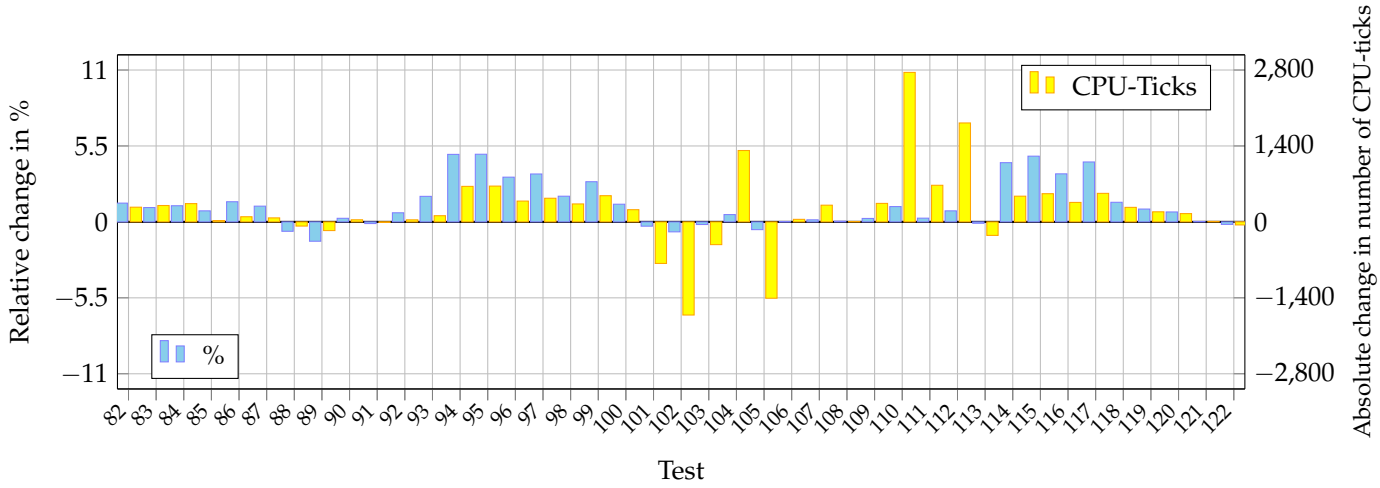


Figure 7.18: Detailed result for Merged Align flags optimization, test 82-122

7.3.2 Sets with small modifications

Table that includes the flags in these test runs.

Flags	7.3.2.1	7.3.2.2	7.3.2.3	7.3.2.4
Original	yes	yes	yes	yes
fthread-jumps	negative	negative	negative	negative
falign-functions	no	no	no	no
falign-jumps	no	no	no	no
falign-loops	no	no	no	no
falign-labels	no	no	no	no
fcaller-saves	yes	no	no	no
fcrossjumping	negative	no	no	no
fcse-follow-jumps	yes	yes	yes	yes
fcse-skip-blocks	yes	no	yes	yes
fdelete-null-pointer-checks	no	no	no	no
fexpensive-optimizations	yes	no	no	yes
fgcse	no	no	no	no
fgcse-lm	no	no	no	no
foptimize-sibling-calls	yes	yes	yes	yes
fpeephole2	no	no	no	no
fregmove	no	no	no	no
freorder-blocks	yes	no	no	no

Table 7.13: List of flags used in the step sets

7.3.2.1 First Set

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	2.08	423	32	2.92	195	63	0.71	233	93	2.77	170
2	2.55	408	33	-0.46	-32	64	0.73	228	94	4.36	584
3	2.14	354	34	-0.58	-41	65	3.30	1479	95	5.55	747
4	1.95	194	35	-0.30	-21	66	3.77	1726	96	3.98	472
5	1.84	224	36	0.33	22	67	6.87	773	97	6.01	755
6	2.29	219	37	1.04	17	68	6.59	806	98	8.81	1568
7	1.36	135	38	3.82	228	69	5.42	665	99	8.47	1407
8	2.28	278	39	3.18	245	70	2.65	128	100	8.04	1412
9	2.54	244	40	4.80	621	71	2.62	123	101	-0.33	-810
10	4.86	247	41	1.22	255	72	3.35	159	102	5.06	12153
11	5.74	290	42	3.26	559	73	0.67	21	103	6.81	15001
12	7.37	1276	43	3.69	519	74	0.54	17	104	4.43	10902
13	-0.50	-163	44	5.45	225	75	3.73	113	105	9.76	24399
14	1.00	203	45	5.54	418	76	-6.31	-520	106	10.04	25007
15	12.35	979	46	3.27	2541	77	4.66	290	107	7.42	15398
16	11.94	949	47	3.59	2786	78	-5.73	-475	108	8.66	1280
17	10.52	833	48	2.52	2042	79	6.63	56	109	2.18	2995
18	0.09	7	49	0.89	277	80	3.73	466	110	10.10	25156
19	0.92	69	50	1.19	387	81	4.16	511	111	0.67	1653
20	3.21	3142	51	1.32	482	82	5.29	1053	112	10.15	23144
21	1.18	23	52	1.65	574	83	3.69	1072	113	10.02	25100
22	1.70	583	53	1.72	514	84	3.60	1038	114	3.98	440
23	1.05	65	54	1.03	333	85	1.99	62	115	3.91	427
24	-0.73	-11	55	0.61	220	86	3.75	244	116	3.43	354
25	6.64	1158	56	0.66	227	87	5.16	335	117	4.30	521
26	6.51	1142	57	2.80	780	88	4.78	538	118	2.54	479
27	3.29	706	58	0.22	65	89	2.53	289	119	0.09	18
28	3.31	698	59	-0.31	-104	90	2.99	441	120	2.63	557
29	2.92	657	60	0.36	115	91	2.11	153	121	4.57	1432
30	-0.46	-96	61	3.41	940	92	2.13	123	122	4.64	1459
31	4.39	230	62	1.33	396						

Table 7.14: Table of detailed values of WCET in percentage for First Set

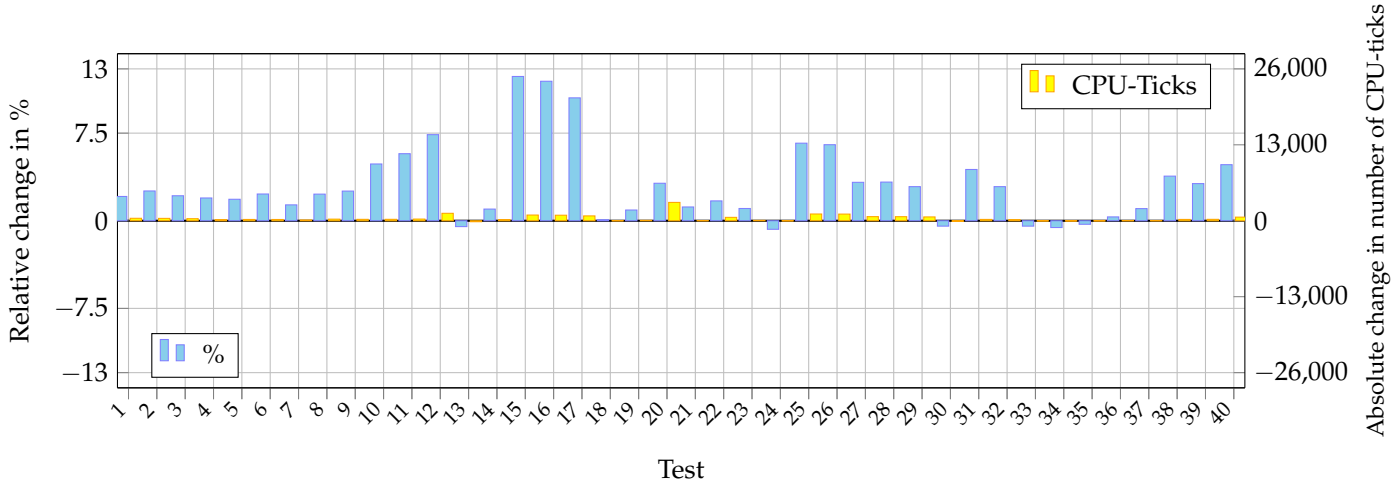


Figure 7.19: Detailed result for First Set, test 1-40

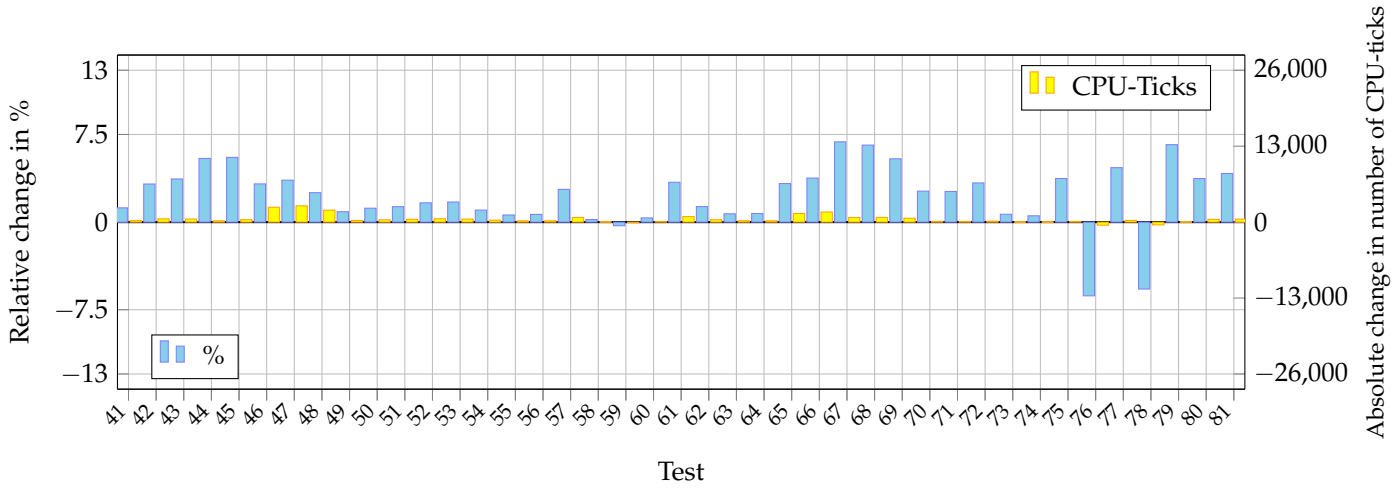


Figure 7.20: Detailed result for First Set, test 41-81

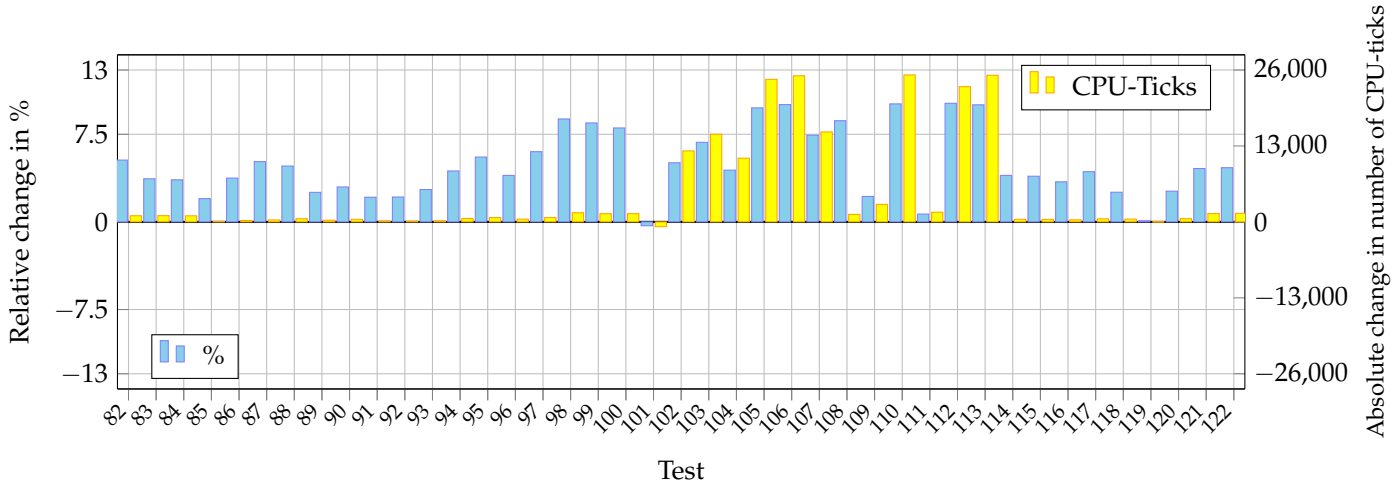


Figure 7.21: Detailed result for First Set, test 82-122

7.3.2.2 Second Set

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	0.86	175	32	5.27	352	63	-0.17	-56	93	2.42	149
2	0.53	85	33	-0.85	-59	64	-0.74	-232	94	5.06	677
3	0.58	96	34	-0.67	-47	65	1.13	507	95	5.29	713
4	3.38	336	35	-0.53	-37	66	1.94	887	96	4.34	515
5	2.29	278	36	-0.29	-19	67	7.54	849	97	5.21	655
6	2.74	262	37	2.44	40	68	7.14	873	98	2.71	483
7	2.65	264	38	-0.25	-15	69	5.59	686	99	2.29	380
8	2.05	250	39	-2.45	-189	70	1.39	67	100	1.97	346
9	2.33	224	40	1.30	168	71	2.47	116	101	0.33	808
10	4.27	217	41	0.48	101	72	3.10	147	102	1.11	2669
11	6.50	328	42	-0.62	-107	73	-0.57	-18	103	0.57	1248
12	3.23	559	43	0.09	13	74	-2.26	-71	104	1.84	4535
13	-1.12	-364	44	5.88	243	75	0.30	9	105	-0.27	-670
14	0.18	37	45	2.00	151	76	3.48	287	106	0.80	1988
15	6.88	545	46	1.96	1518	77	2.39	149	107	-0.37	-773
16	8.58	682	47	0.50	386	78	1.95	162	108	-0.12	-17
17	7.54	597	48	1.77	1435	79	3.20	27	109	-0.10	-141
18	-0.16	-12	49	-2.09	-649	80	2.86	357	110	0.86	2154
19	-1.53	-115	50	1.16	380	81	2.75	338	111	-1.39	-3438
20	0.72	706	51	0.30	108	82	2.60	518	112	0.11	259
21	-2.21	-43	52	0.95	329	83	1.36	394	113	0.65	1619
22	0.31	05	53	-1.15	-343	84	1.68	483	114	4.77	528
23	0.37	23	54	-0.43	-139	85	3.15	98	115	4.08	446
24	-8.63	-130	55	-0.33	-121	86	2.12	138	116	4.37	451
25	5.87	1023	56	-0.30	-104	87	3.99	259	117	3.37	409
26	4.24	750	57	0.13	35	88	5.07	571	118	2.84	535
27	5.12	1098	58	-0.13	-38	89	3.08	352	119	0.52	105
28	4.50	948	59	-0.86	-293	90	1.78	263	120	3.12	661
29	10.28	2312	60	0.64	201	91	2.28	165	121	0.80	251
30	-0.02	-5	61	-0.16	-44	92	1.72	99	122	0.56	175
31	6.63	347	62	0.87	259						

Table 7.15: Table of detailed values of WCET in percentage for Second Set

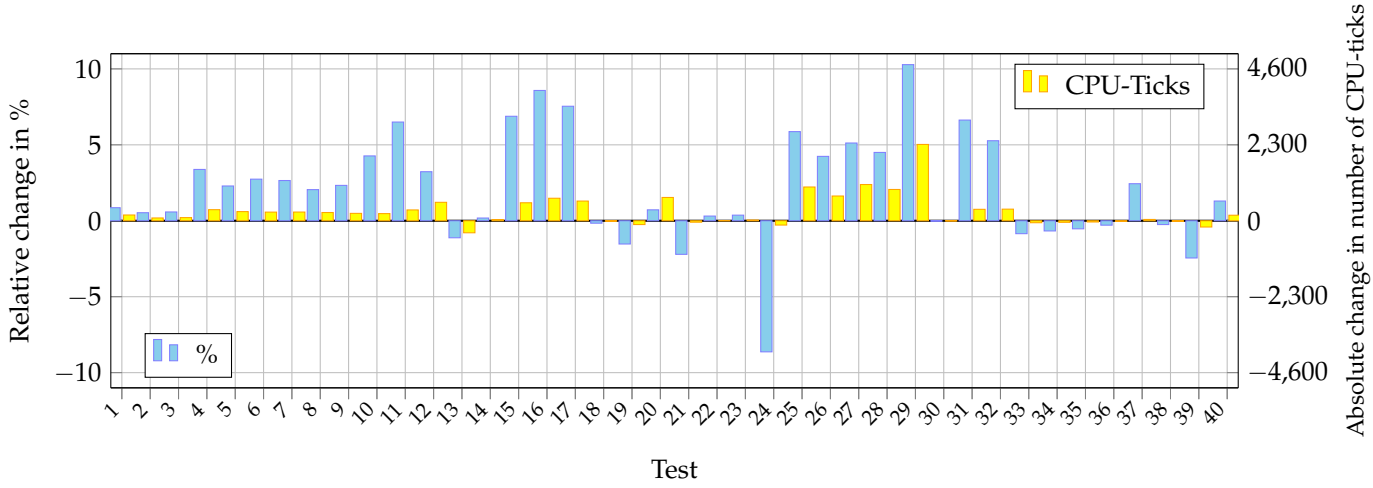


Figure 7.22: Detailed result for Second Set optimization, test 1-40

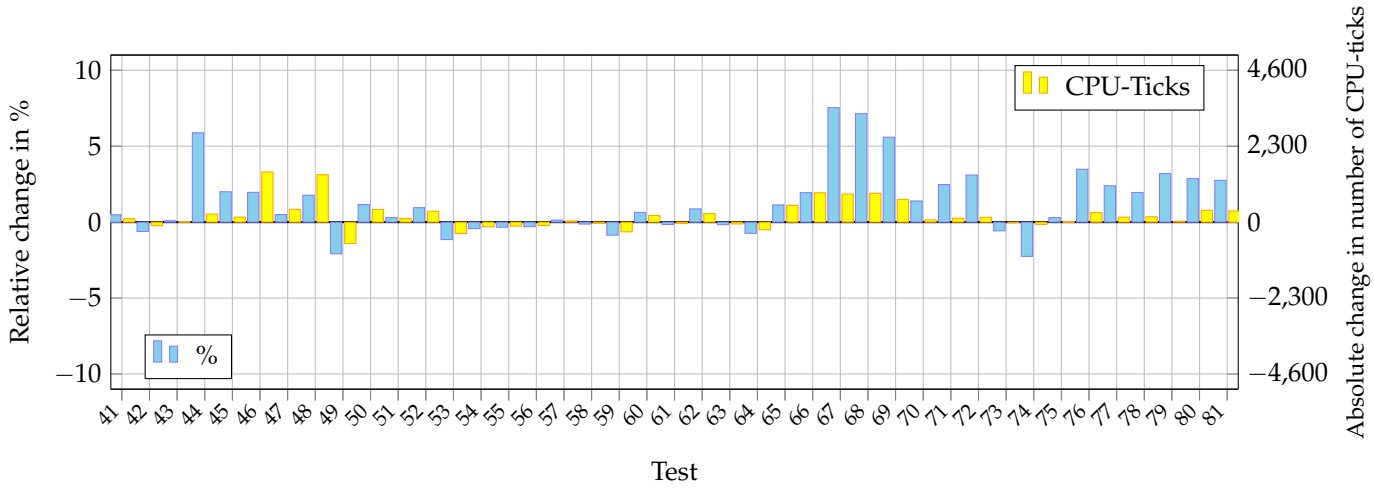


Figure 7.23: Detailed result for Second Set optimization, test 41-81

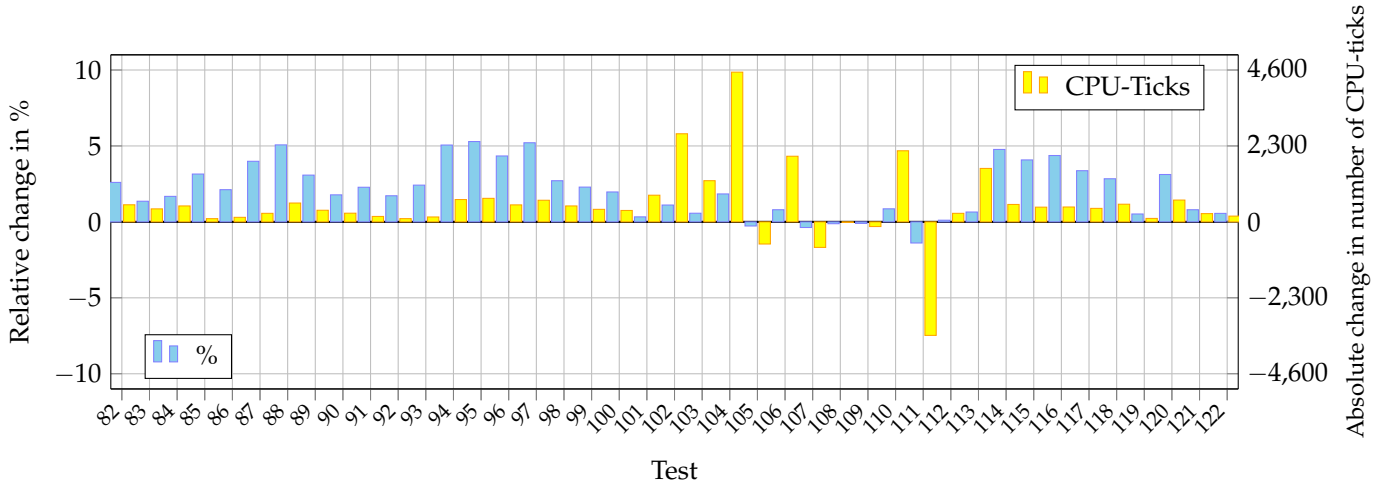


Figure 7.24: Detailed result for Second Set optimization, test 82-122

7.3.2.3 Third Set

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	1.71	348	32	2.04	136	63	-0.66	-218	93	2.47	152
2	1.25	201	33	-0.85	-59	64	-0.28	-86	94	5.15	689
3	1.27	209	34	-0.81	-57	65	1.23	549	95	6.05	815
4	-0.21	-21	35	-0.58	-40	66	1.76	804	96	2.68	318
5	0.35	43	36	-0.33	-22	67	5.30	596	97	2.63	331
6	0.10	10	37	-1.10	-18	68	3.99	488	98	0.05	9
7	-0.53	-53	38	2.66	159	69	3.44	422	99	2.15	358
8	0.27	33	39	2.08	160	70	-0.37	-18	100	0.99	174
9	-0.02	-2	40	1.53	198	71	1.15	54	101	0.55	1340
10	3.39	172	41	0.91	191	72	0.88	42	102	1.00	2391
11	6.04	305	42	0.26	45	73	-1.18	-37	103	0.55	1206
12	-1.01	-175	43	1.15	164	74	-0.73	-23	104	0.04	105
13	-1.76	-570	44	5.59	231	75	0.50	15	105	-0.62	-1559
14	-0.07	-15	45	2.15	162	76	1.90	157	106	0.04	112
15	5.90	468	46	2.35	1824	77	2.99	186	107	-0.34	-704
16	5.69	452	47	2.97	2301	78	1.98	164	108	2.60	384
17	6.25	495	48	0.72	587	79	1.54	13	109	-0.71	-978
18	-0.72	-54	49	-1.86	-578	80	3.97	496	110	0.14	344
19	-1.51	-114	50	1.02	333	81	2.77	340	111	-1.97	-4845
20	1.87	1837	51	0.28	104	82	2.37	472	112	-0.83	-1899
21	-2.51	-49	52	0.87	302	83	1.47	427	113	0.20	511
22	0.52	178	53	-1.91	-572	84	1.79	516	114	2.12	235
23	0.29	18	54	0.39	125	85	3.72	116	115	2.96	323
24	-5.84	-88	55	-0.31	-111	86	-1.57	-102	116	3.12	322
25	4.99	870	56	0.13	46	87	-1.59	-103	117	3.50	424
26	2.40	421	57	-0.67	-187	88	2.44	275	118	2.85	538
27	7.21	1547	58	-0.23	-68	89	1.87	214	119	0.55	110
28	6.06	1277	59	-0.58	-198	90	0.57	84	120	-1.17	-249
29	1.31	295	60	-0.41	-131	91	1.74	126	121	1.16	365
30	-1.74	-367	61	0.63	173	92	2.31	133	122	0.96	303
31	4.32	226	62	0.78	233						

Table 7.16: Table of detailed values of WCET in percentage for Third Set

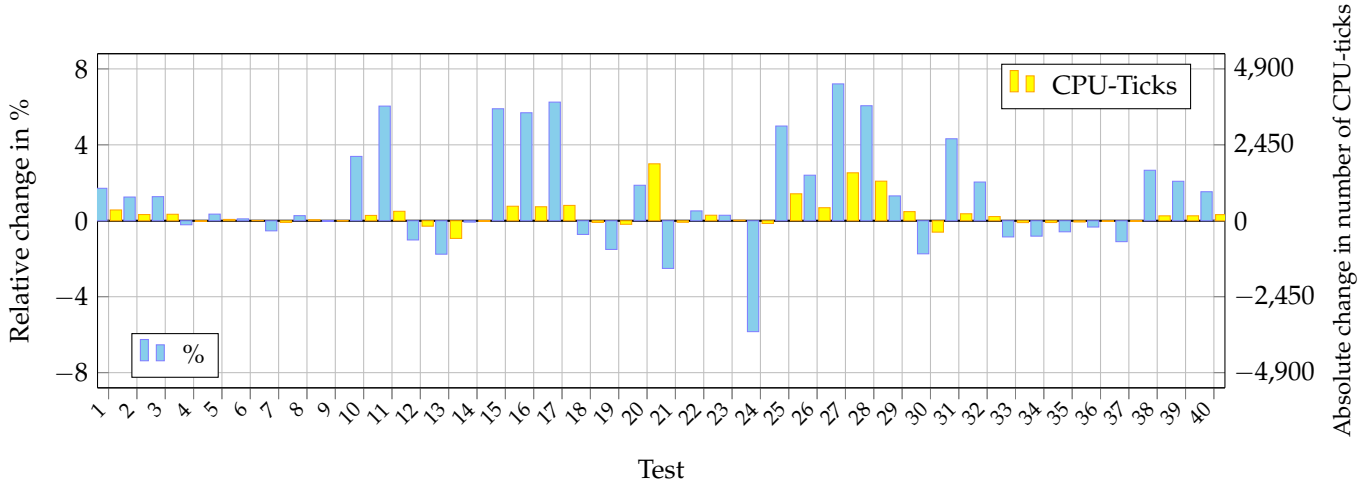


Figure 7.25: Detailed result for Third Set optimization, test 1-40

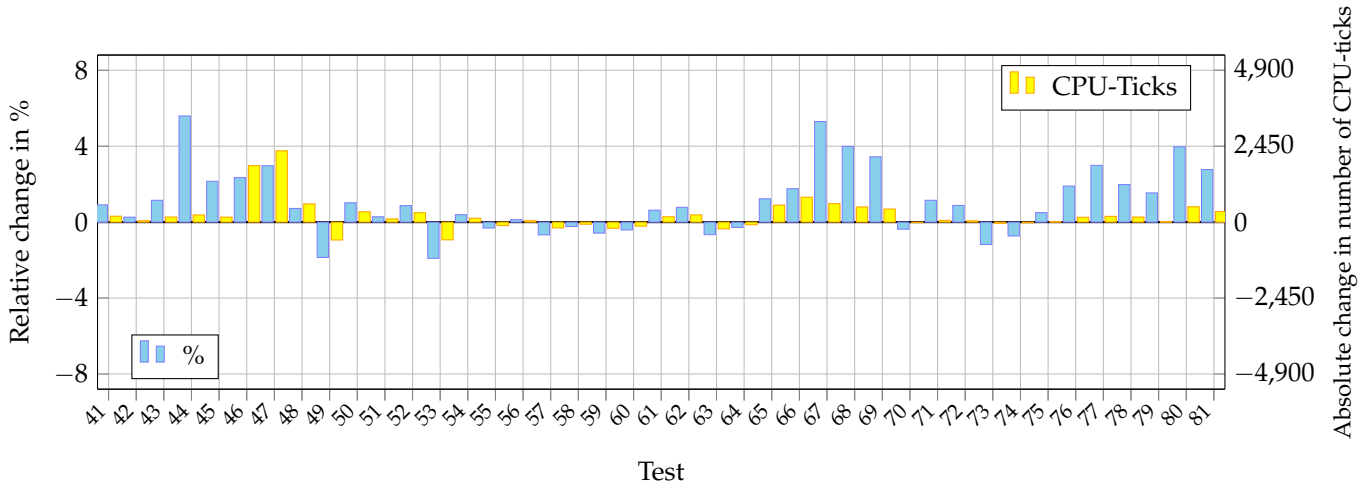


Figure 7.26: Detailed result for Third Set optimization, test 41-81

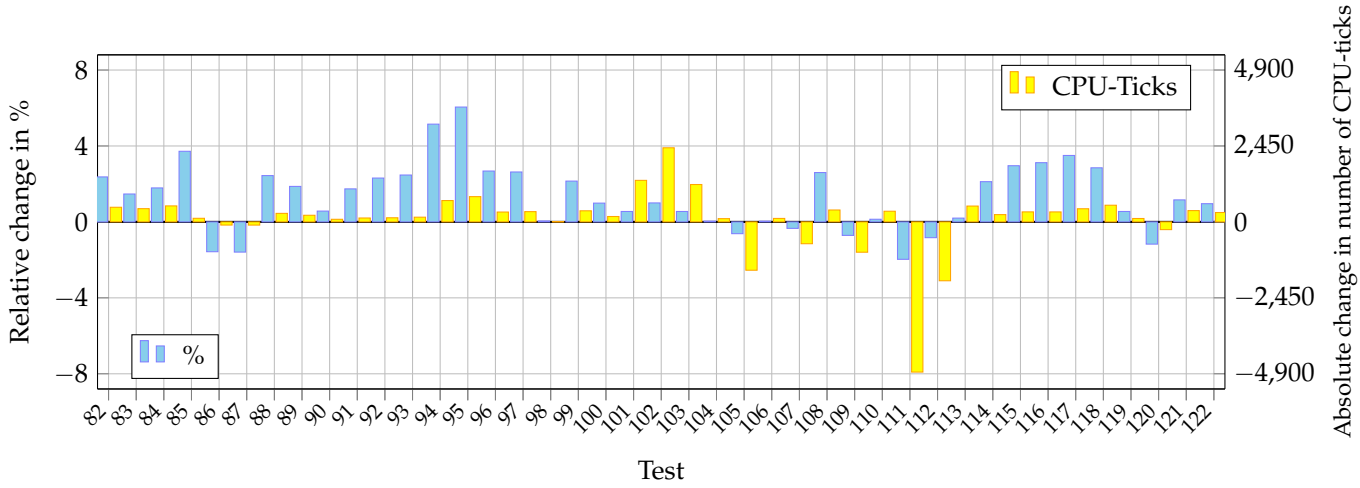


Figure 7.27: Detailed result for Third Set optimization, test 82-122

7.3.2.4 Fourth Set

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	0.26	52	32	-0.46	-31	63	-1.71	-565	93	0.54	33
2	-1.60	-256	33	-1.72	-120	64	-0.73	-229	94	1.62	217
3	-1.51	-250	34	-1.51	-106	65	1.90	851	95	3.35	451
4	0.36	36	35	-1.25	-87	66	3.09	1414	96	1.07	127
5	-0.12	-15	36	-1.03	-68	67	3.12	351	97	1.81	228
6	0.40	38	37	0.61	10	68	1.93	236	98	-0.08	-15
7	0.32	32	38	2.30	137	69	1.11	136	99	1.68	279
8	0.11	14	39	0.64	49	70	-9.62	-464	100	1.44	252
9	0.44	42	40	-2.60	-336	71	-8.59	-404	101	0.32	795
10	-0.18	-9	41	-1.36	-285	72	-6.72	-319	102	0.25	612
11	-3.51	-177	42	-1.20	-205	73	-9.26	-310	103	-0.07	-150
12	0.38	66	43	-2.76	-395	74	-11.12	-349	104	0.24	597
13	-1.16	-376	44	1.26	52	75	-6.67	-202	105	-0.82	-2053
14	0.20	41	45	2.03	153	76	1.58	130	106	-0.14	-340
15	3.17	251	46	0.73	568	77	0.72	45	107	-0.64	-1331
16	2.44	194	47	1.25	970	78	1.13	94	108	1.05	155
17	3.04	241	48	-0.72	-582	79	9.11	77	109	-0.06	-87
18	1.69	126	49	-3.92	-1221	80	-2.34	-292	110	0.54	1351
19	-0.16	-12	50	-0.15	-50	81	-2.84	-349	111	-1.49	-3665
20	0.84	826	51	-0.73	-268	82	2.68	533	112	-0.64	-1468
21	-0.97	-19	52	-0.36	-125	83	1.45	421	113	0.37	917
22	0.43	149	53	-1.45	-432	84	1.93	557	114	2.69	297
23	1.32	82	54	0.68	219	85	-1.38	-43	115	2.09	228
24	-10.02	-151	55	-0.13	-47	86	-2.29	-149	116	1.80	186
25	4.98	868	56	0.10	33	87	-2.83	-184	117	2.79	339
26	2.88	505	57	-1.00	-279	88	2.44	274	118	-1.35	-255
27	3.19	684	58	-0.40	-119	89	0.63	72	119	-1.60	-322
28	2.58	544	59	-0.18	-62	90	0.12	18	120	0.16	33
29	7.71	1733	60	-0.12	-37	91	-0.11	-8	121	-0.36	-113
30	-2.29	-484	61	0.95	261	92	1.49	86	122	-0.31	-99
31	0.61	32	62	0.74	221						

Table 7.17: Table of detailed values of WCET in percentage for Fourth Set

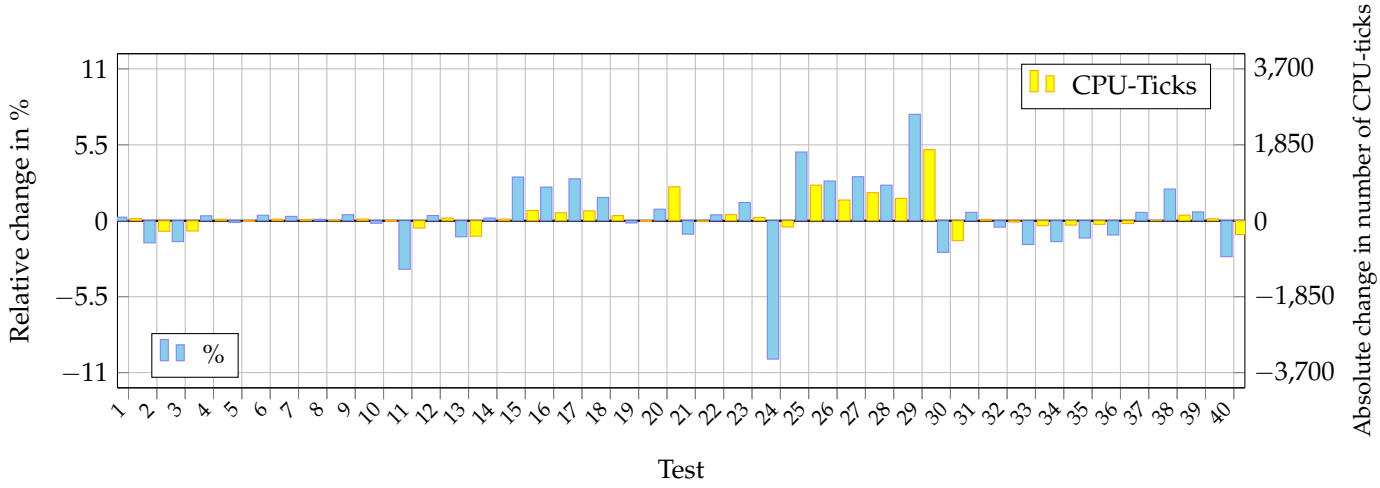


Figure 7.28: Detailed result for Fourth Set optimization, test 1-40

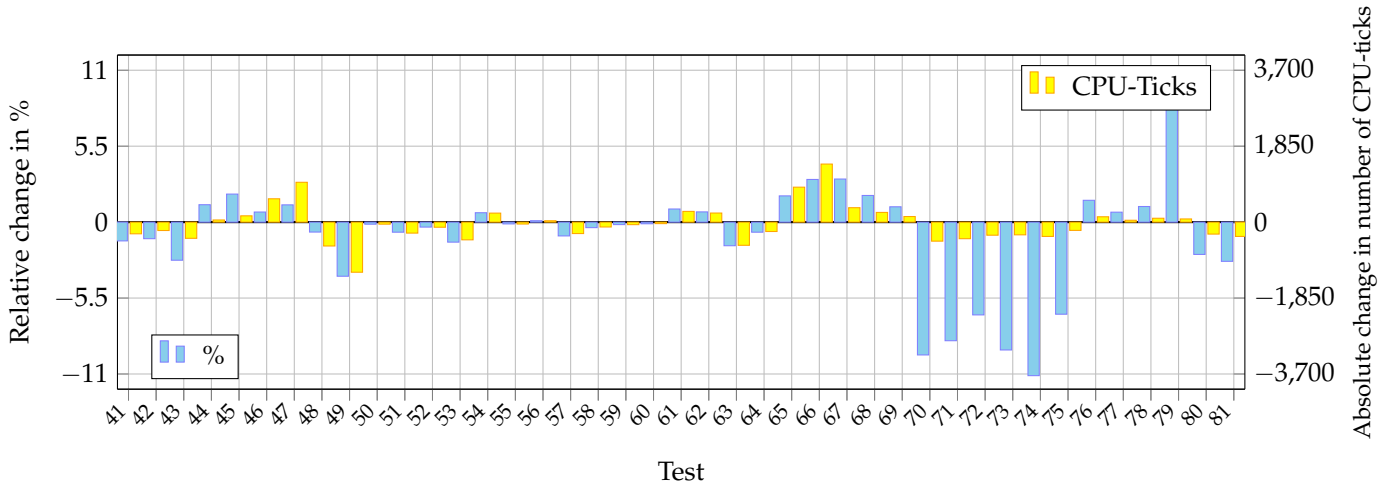


Figure 7.29: Detailed result for Fourth Set optimization, test 41-81

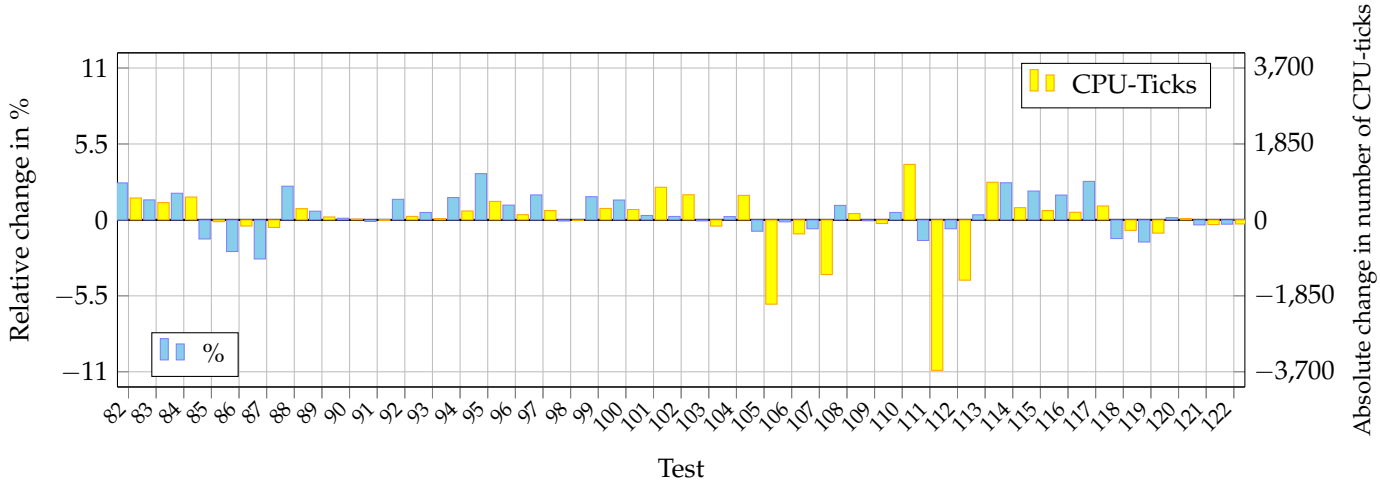


Figure 7.30: Detailed result for Fourth Set optimization, test 82-122

7.3.3 The O(ptimization) groups

Table that includes the flags in these test runs.

Flags	7.3.3.1	7.3.3.2
Original	yes	yes
fcaller-saves	yes	yes
fcrossjumping	negative	negative
fcse-follow-jumps	yes	yes
fcse-skip-blocks	yes	yes
fdelete-null-pointer-checks	yes	yes
fexpensive-optimizations	yes	yes
fgcse	yes	yes
fgcse-lm	yes	yes
foptimize-sibling-calls	yes	yes
fpeephole2	yes	yes
fregmove	yes	yes
freorder-blocks	yes	yes
freorder-functions	yes	yes
frerun-cse-after-loop	yes	yes
fsched-interblock	yes	yes
fsched-spec	yes	yes
fschedule-insns	yes	yes
fschedule-insns2	yes	yes
fstrict-aliasing	yes	yes
fstrict-overflow	yes	yes
ftree-pre	yes	yes
ftree-vrp	yes	yes
finline-functions	no	yes
funswitch-loops	no	yes
fpredictive-commoning	no	yes
fgcse-after-reload	no	yes
ftree-vectorize	no	yes

Table 7.18: List of flags used in the modified O groups

7.3.3.1 -O2

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	-0.64	-130	32	1.92	128	63	-2.92	-964	93	-2.49	-153
2	-2.92	-468	33	-0.20	-14	64	-2.18	-681	94	4.24	567
3	-2.76	-455	34	-0.01	-1	65	3.31	1483	95	4.48	604
4	-4.08	-405	35	0.22	15	66	4.62	2115	96	0.51	61
5	-1.45	-176	36	0.09	6	67	3.14	353	97	1.86	234
6	-3.01	-288	37	-0.31	-5	68	2.93	358	98	2.97	529
7	-3.13	-312	38	2.33	139	69	1.85	227	99	3.88	645
8	-2.04	-248	39	4.04	311	70	-2.09	-101	100	2.84	499
9	-3.66	-351	40	2.33	301	71	-2.64	-124	101	-6.41	-15718
10	4.35	221	41	-0.18	-37	72	-0.76	-36	102	-.145	-3474
11	6.02	304	42	1.76	301	73	-8.49	-267	103	-0.56	-1236
12	7.79	1350	43	3.32	475	74	-9.56	-300	104	-0.44	-1084
13	-1.11	-361	44	2.98	123	75	-5.48	-166	105	1.89	4736
14	0.32	65	45	1.67	126	76	1.18	97	106	2.20	5470
15	10.55	836	46	2.43	1884	77	-1.08	-67	107	2.41	5009
16	11.02	876	47	1.42	1104	78	1.23	102	108	4.35	643
17	9.56	757	48	2.69	2178	79	-2.60	-22	109	-8.63	-11852
18	0.97	72	49	-2.33	-725	80	5.75	717	110	2.36	5887
19	-0.20	-15	50	-0.87	-283	81	4.70	577	111	-4.38	-10802
20	1.28	1255	51	-1.95	-714	82	4.71	936	112	5.02	11443
21	0.05	1	52	-1.04	-360	83	2.55	740	113	2.06	5163
22	2.06	706	53	1.03	308	84	2.22	639	114	-0.39	-43
23	4.53	-281	54	-0.82	-263	85	1.32	41	115	-0.09	-10
24	-1.39	-21	55	-2.36	-854	86	2.47	161	116	-0.27	-28
25	1.27	222	56	-1.38	-472	87	1.63	106	117	-0.66	-80
26	1.82	320	57	-0.22	-61	88	-0.62	-70	118	2.85	537
27	5.90	1265	58	-2.54	-762	89	-1.98	-227	119	2.00	402
28	6.03	1271	59	-2.94	-1000	90	1.50	221	120	0.85	181
29	4.43	997	60	-1.47	-466	91	-3.49	-253	121	4.61	1445
30	-1.92	-405	61	0.33	90	92	0.66	38	122	4.09	1287
31	4.22	221	62	-2.00	-596						

Table 7.19: Table of detailed values of WCET in percentage for -O2

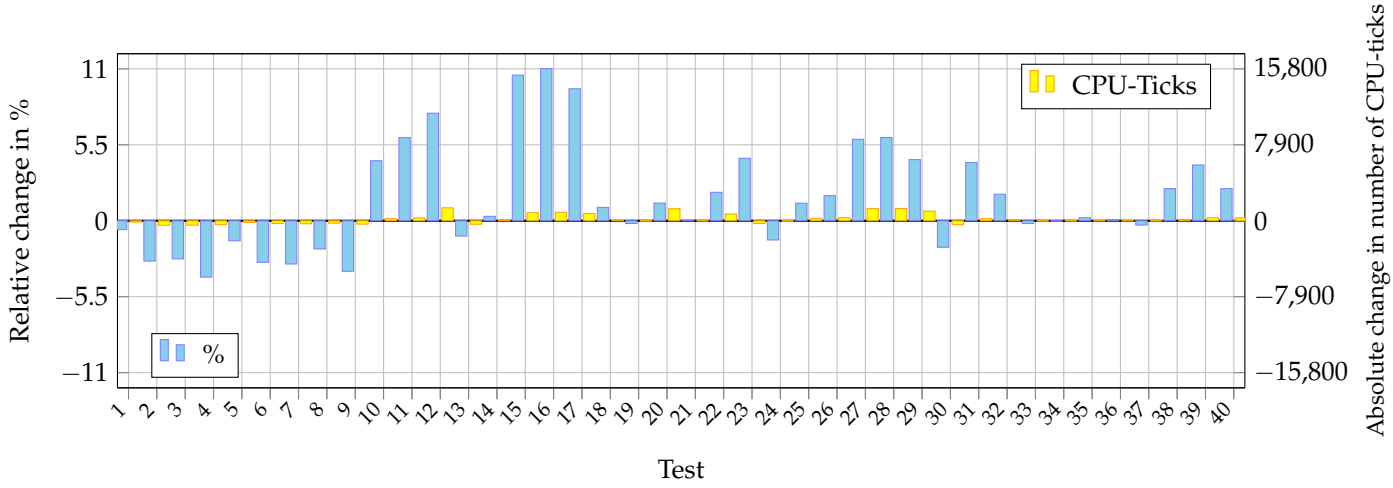


Figure 7.31: Detailed result for -O2 optimization, test 1-40

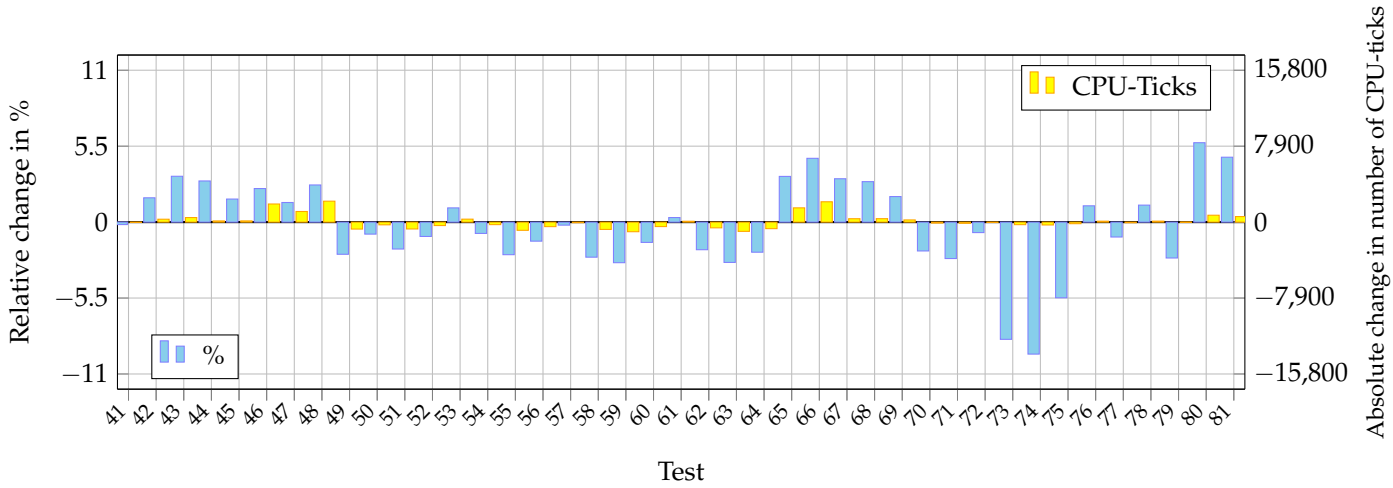


Figure 7.32: Detailed result for -O2 optimization, test 41-81

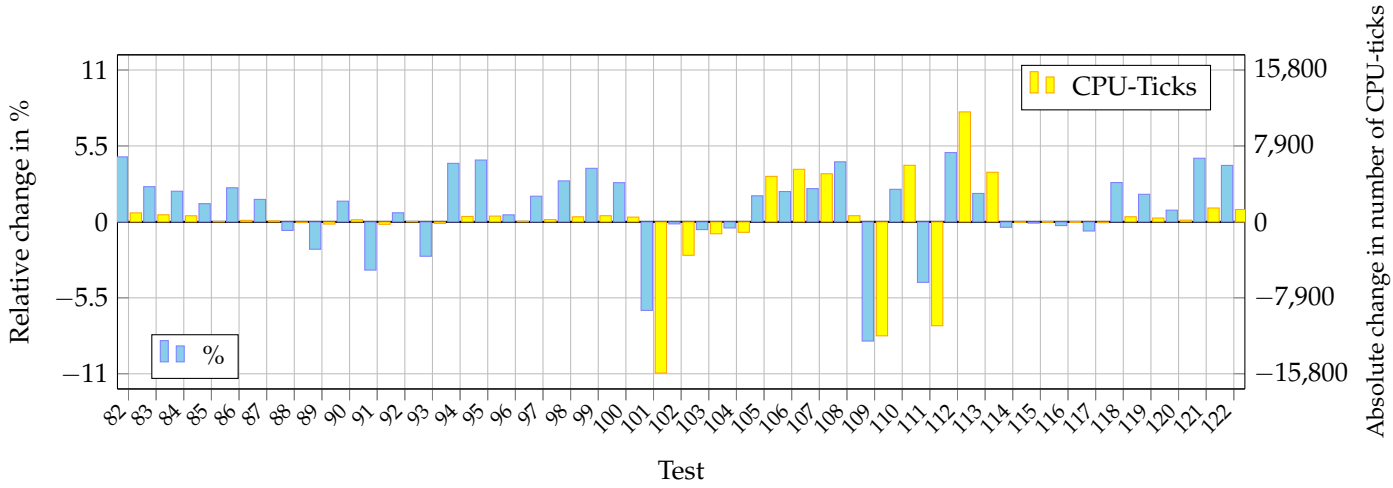


Figure 7.33: Detailed result for -O2 optimization, test 82-122

7.3.3.2 -O3

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	-4.52	-922	32	1.98	132	63	-2.13	-701	93	-6.65	-409
2	-5.98	-958	33	2.52	176	64	-0.58	-181	94	1.00	134
3	-5.73	-946	34	2.69	189	65	1.35	606	95	1.72	232
4	-9.04	-898	35	2.99	208	66	2.36	1078	96	0.20	24
5	-6.37	-774	36	3.17	210	67	6.04	680	97	2.44	307
6	-8.38	-801	37	-1.77	-29	68	4.38	535	98	7.06	1257
7	-8.88	-884	38	2.18	130	69	3.43	421	99	8.25	1370
8	-6.30	-767	39	2.77	213	70	-5.06	-244	100	8.44	1481
9	-8.71	-836	40	4.04	522	71	-5.42	-255	101	-6.77	-16612
10	-4.76	-242	41	0.95	199	72	-4.82	-229	102	-2.44	-5846
11	-4.20	-212	42	-0.44	-76	73	-2.58	-81	103	0.63	1389
12	6.29	1089	43	3.10	444	74	-1.85	-58	104	0.06	154
13	-0.88	-286	44	7.48	309	75	3.33	101	105	0.94	2345
14	1.59	321	45	2.23	168	76	4.57	377	106	1.46	3648
15	10.63	843	46	-8.10	-6285	77	3.65	227	107	-0.99	-2056
16	9.54	758	47	-8.48	-6571	78	4.08	338	108	8.76	1294
17	7.74	613	48	-9.10	-7367	79	0.59	5	109	-7.43	-10211
18	0.66	49	49	-1.86	-578	80	1.63	204	110	1.68	4196
19	-0.09	-7	50	-0.19	-61	81	0.77	95	111	-4.45	-10963
20	0.24	240	51	-1.36	-497	82	4.82	959	112	3.24	7390
21	0.72	14	52	-0.35	-120	83	3.08	894	113	1.49	3734
22	-16.01	-5484	53	0.92	274	84	3.56	1024	114	5.09	563
23	-8.99	-558	54	-1.19	-382	85	4.33	135	115	6.59	719
24	-2.79	-42	55	-2.25	-811	86	-0.02	-1	116	5.99	618
25	0.40	70	56	-1.06	-362	87	0.77	50	117	5.47	663
26	0.05	8	57	1.31	364	88	-1.31	-147	118	-1.14	-215
27	0.85	183	58	-1.70	-511	89	-2.25	-257	119	-0.61	-123
28	0.43	90	59	-2.47	-838	90	1.92	284	120	-2.81	-596
29	3.25	731	60	-0.61	-192	91	-7.37	-534	121	4.01	1257
30	-1.23	-259	61	3.28	903	92	-5.36	-309	122	3.71	1166
31	5.86	307	62	-0.08	-23						

Table 7.20: Table of detailed values of WCET in percentage for -O3

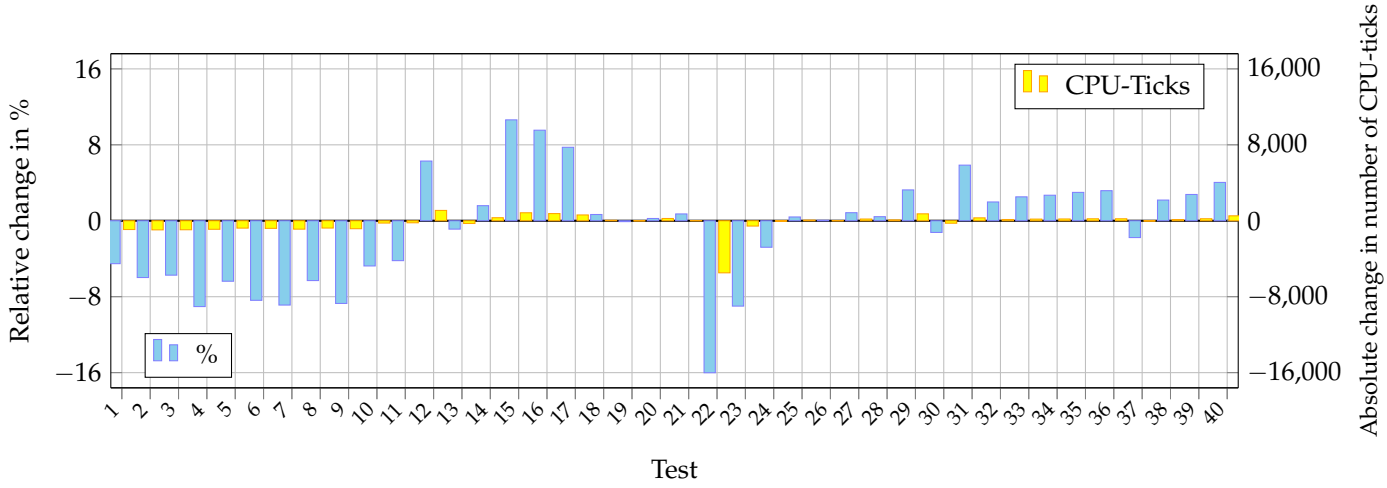


Figure 7.34: Detailed result for -O3 optimization, test 1-40

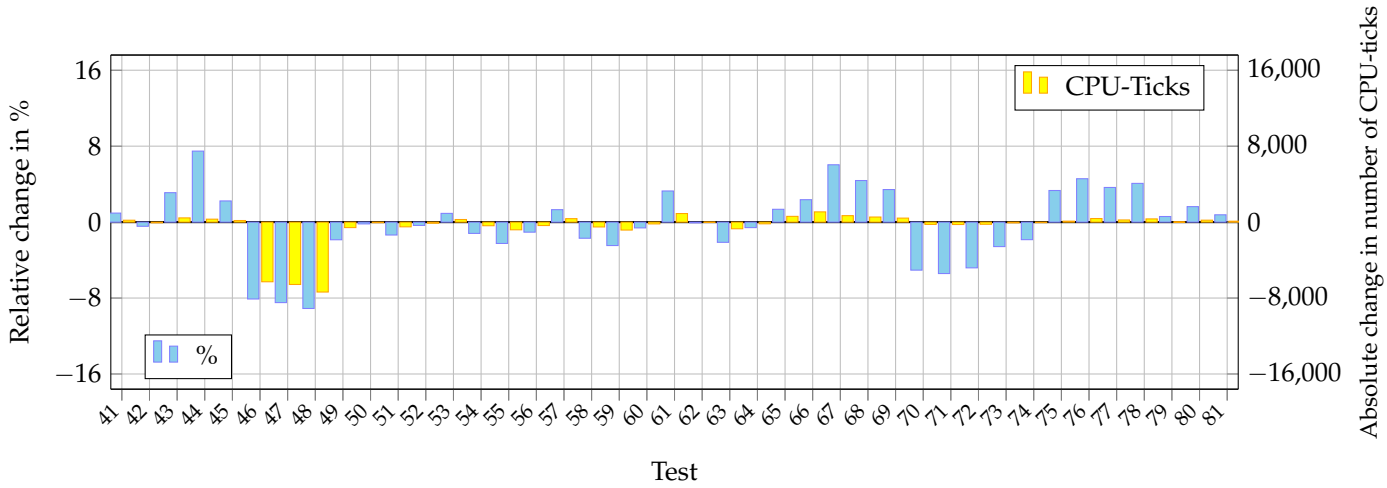


Figure 7.35: Detailed result for -O3 optimization, test 41-81

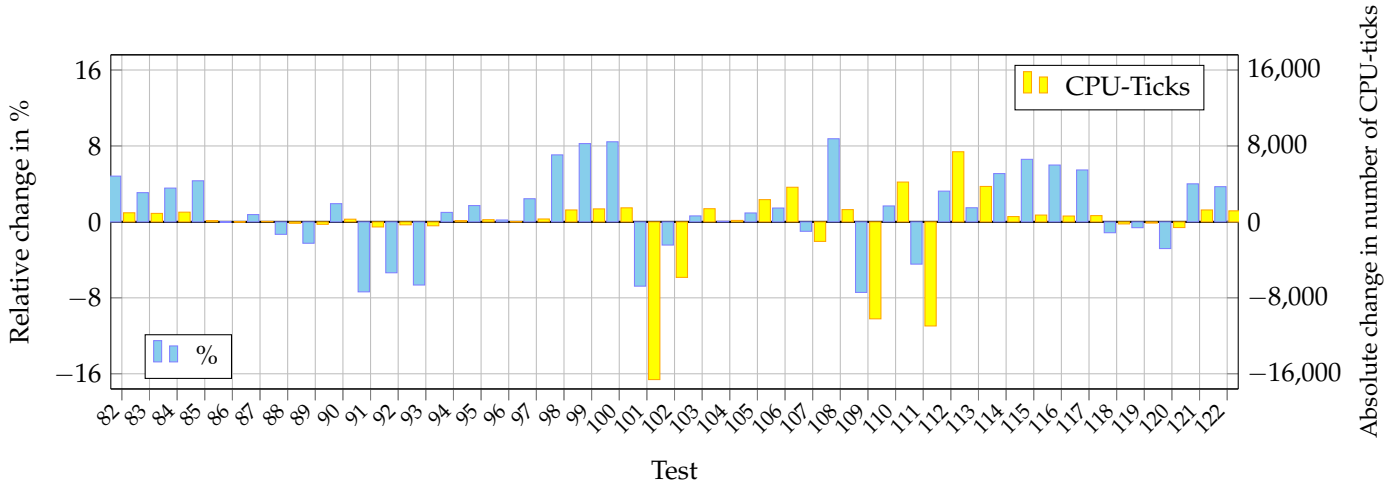


Figure 7.36: Detailed result for -O3 optimization, test 82-122

7.3.4 Other test runs

Table that includes the flags in these test runs.

Flags	7.3.4.3	7.3.4.2	7.3.4.1
Original	yes	yes	yes
falign-functions	no	no	no
falign-jumps	no	no	no
falign-loops	no	no	no
falign-labels	no	no	no
fcaller-saves	no	no	no
fcrossjumping	no	no	no
fcse-follow-jumps	no	no	no
fcse-skip-blocks	no	no	no
fdelete-null-pointer-checks	no	no	no
fexpensive-optimizations	yes	no	no
fgcse	no	no	no
fgcse-lm	no	no	no
foptimize-sibling-calls	yes	yes	no
fpeephole2	no	no	no
fregmove	no	no	no
freorder-blocks	no	yes	no

Table 7.21: Example of a table with the standalone sets

7.3.4.1 No Sibling Optimization

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	0.38	78	32	4.18	279	63	0.86	285	93	2.12	130
2	4.20	673	33	1.19	83	64	0.47	148	94	5.18	693
3	4.11	679	34	0.80	56	65	1.45	649	95	5.89	793
4	4.15	412	35	1.47	102	66	1.60	733	96	4.04	480
5	3.70	450	36	0.94	62	67	6.23	701	97	4.06	510
6	3.21	307	37	3.11	51	68	4.38	353	98	4.34	772
7	3.99	397	38	4.12	246	69	4.48	550	99	3.71	616
8	3.92	477	39	2.73	210	70	-0.83	-40	100	1.63	286
9	2.98	286	40	-0.04	-5	71	1.81	85	101	0.72	1757
10	5.67	288	41	0.30	62	72	1.47	70	102	2.37	5690
11	7.53	380	42	-0.95	-163	73	-3.24	-102	103	2.74	6025
12	4.15	718	43	2.04	292	74	-3.15	-99	104	3.79	9342
13	0.10	32	44	8.42	348	75	-1.75	-53	105	2.44	6092
14	-0.07	-14	45	1.55	117	76	3.52	290	106	3.05	7607
15	6.75	535	46	1.52	1177	77	5.54	339	107	0.72	1493
16	6.38	507	47	1.64	1268	78	3.44	285	108	2.04	302
17	6.88	545	48	1.05	847	79	7.34	62	109	0.42	582
18	0.90	67	49	0.58	181	80	3.12	389	110	2.53	6292
19	1.07	81	50	1.15	376	81	2.06	253	111	2.44	6021
20	3.47	3398	51	0.72	264	82	3.39	675	112	3.45	7866
21	4.05	79	52	1.13	394	83	1.97	571	113	2.24	5623
22	1.52	522	53	2.47	738	84	2.11	607	114	3.98	440
23	2.64	164	54	0.48	155	85	4.82	150	115	4.26	465
24	-2.39	-36	55	0.35	128	86	4.13	269	116	3.01	311
25	3.47	605	56	0.05	16	87	4.85	315	117	4.21	511
26	3.21	564	57	2.41	673	88	2.95	332	118	3.58	676
27	5.81	1246	58	0.78	234	89	2.47	282	119	1.76	354
28	6.99	1473	59	0.17	59	90	1.05	155	120	4.34	920
29	0.92	206	60	1.45	458	91	2.17	157	121	2.86	895
30	1.79	377	61	4.12	1134	92	2.64	152	122	2.30	722
31	6.05	317	62	0.95	283						

Table 7.22: Table of detailed values of WCET in percentage for No Sibling

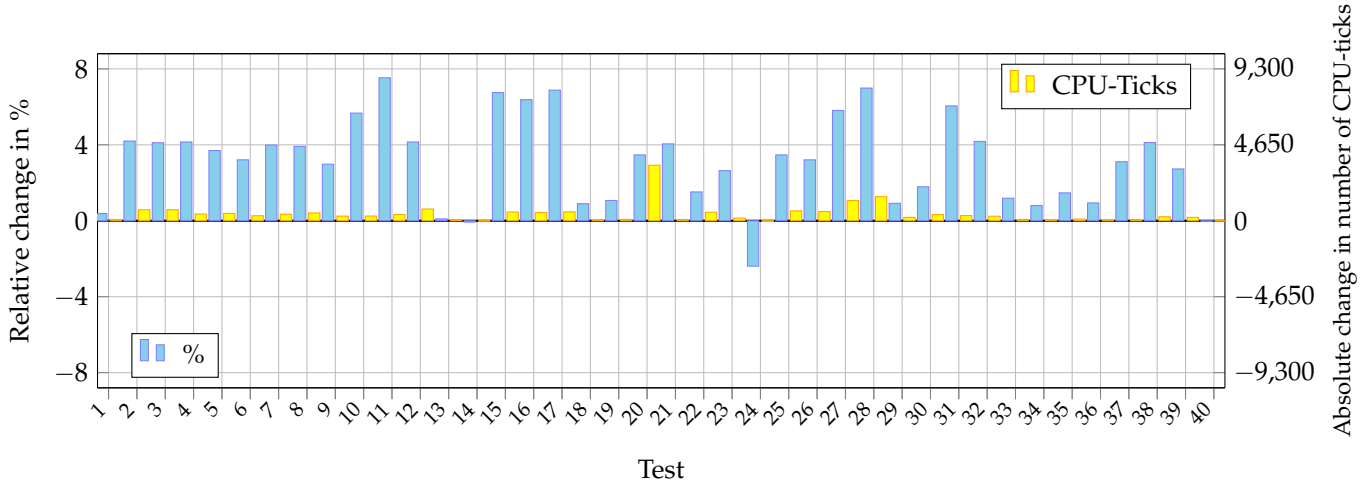


Figure 7.37: Detailed result for No Sibling optimization, test 1-40

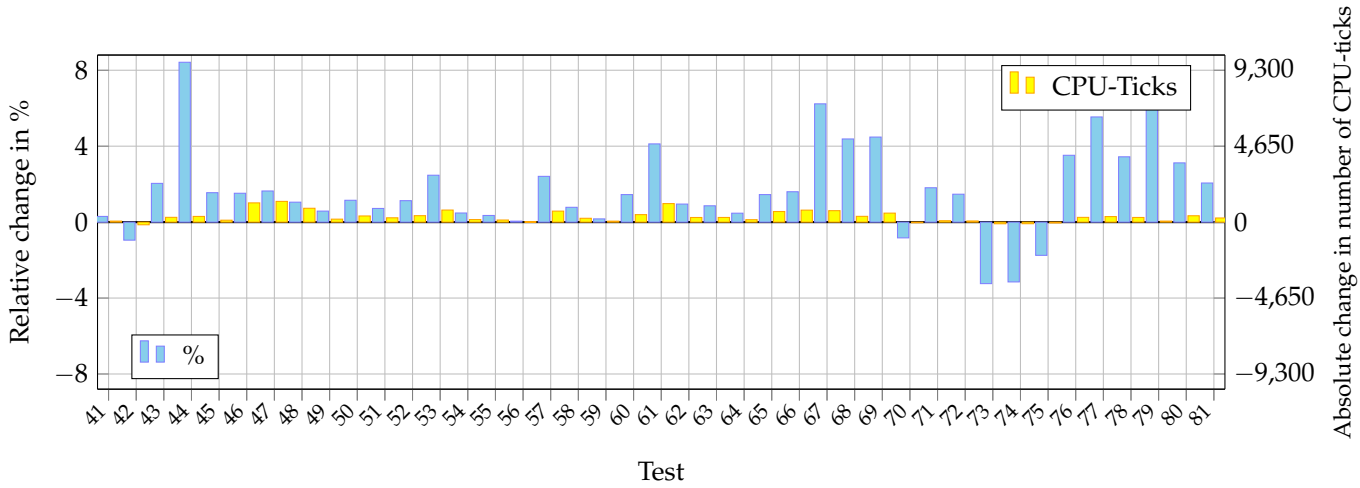


Figure 7.38: Detailed result for No Sibling optimization, test 41-81

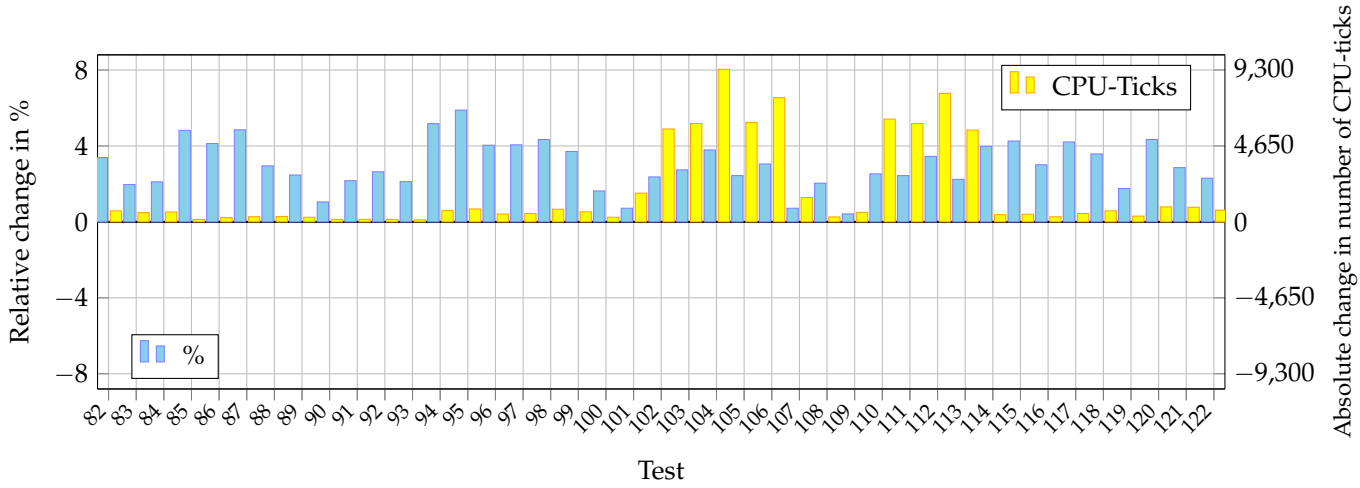


Figure 7.39: Detailed result for No Sibling optimization, test 82-122

7.3.4.2 Reorder Blocks

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	2.60	529	32	3.77	252	63	0.53	174	93	2.52	155
2	1.88	302	33	1.20	84	64	0.05	15	94	5.65	756
3	1.73	286	34	1.47	103	65	-2.17	-974	95	5.65	761
4	2.77	275	35	1.58	110	66	-1.84	-840	96	2.65	314
5	3.20	389	36	0.89	59	67	5.18	583	97	3.97	499
6	2.76	264	37	1.53	25	68	4.53	554	98	7.27	1295
7	3.15	314	38	4.39	262	69	3.93	482	99	7.76	1290
8	3.33	405	39	2.79	215	70	-7.65	-369	100	7.38	1295
9	2.36	227	40	4.04	522	71	-8.36	-393	101	-0.12	-289
10	3.57	181	41	1.22	257	72	-6.74	-320	102	7.71	18520
11	7.23	365	42	3.29	564	73	-3.69	-116	103	7.25	15959
12	8.05	1394	43	5.59	800	74	-3.95	-124	104	8.59	21155
13	-0.49	-159	44	9.44	390	75	-1.06	-32	105	11.45	28625
14	1.20	242	45	4.90	370	76	-8.10	-668	106	11.84	29482
15	8.72	691	46	3.08	2393	77	3.54	220	107	8.64	17934
16	9.28	738	47	0.74	570	78	-5.56	-461	108	7.93	1172
17	7.39	585	48	-0.13	-106	79	0.83	7	109	2.04	2808
18	0.76	57	49	-0.24	-74	80	5.36	669	110	12.01	29912
19	0.03	2	50	1.20	393	81	5.07	622	111	4.54	11196
20	2.41	2359	51	0.67	244	82	5.27	1049	112	11.51	26239
21	1.28	25	52	0.71	246	83	3.45	1002	113	12.83	32143
22	2.24	767	53	0.09	26	84	3.66	1055	114	3.24	358
23	1.84	114	54	0.61	197	85	0.16	5	115	1.84	201
24	0.66	10	55	0.33	118	86	-2.55	-166	116	1.05	108
25	2.48	433	56	0.29	99	87	-1.85	-120	117	2.14	259
26	2.75	483	57	1.80	502	88	3.96	446	118	3.36	634
27	5.37	1152	58	-0.04	-11	89	2.70	309	119	-0.69	-139
28	5.84	1232	59	-0.82	-279	90	-0.46	-68	120	-0.78	-166
29	-0.32	-72	60	1.36	430	91	1.95	141	121	4.96	1554
30	-1.08	-228	61	5.08	1398	92	2.41	139	122	4.99	1570
31	6.25	327	62	1.99	592						

Table 7.23: Table of detailed values of WCET in percentage for Reorder Blocks

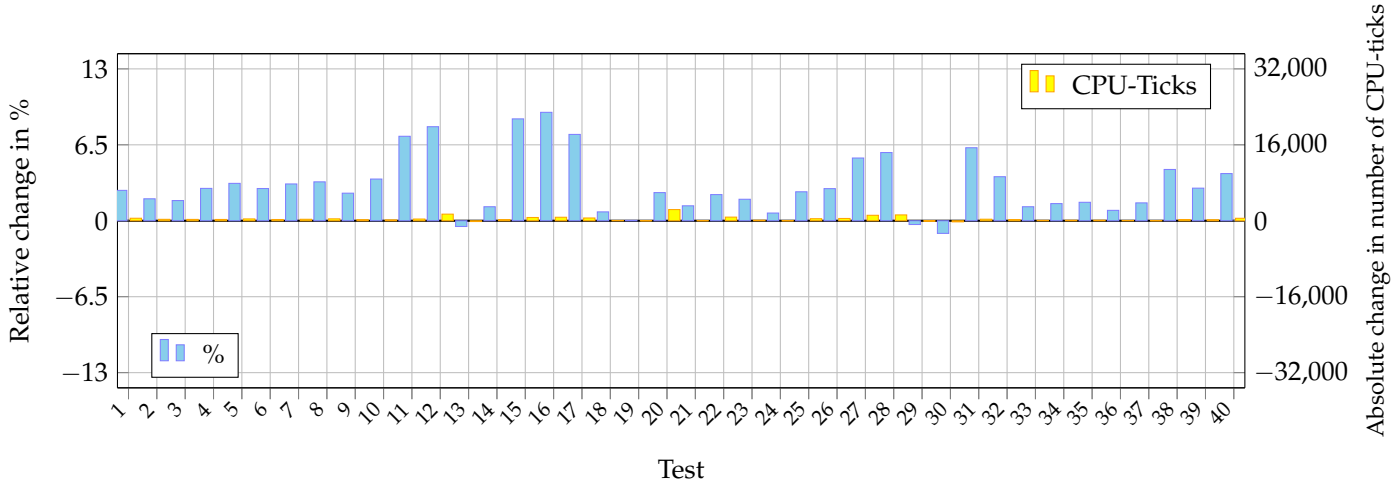


Figure 7.40: Detailed result for Reorder Blocks optimization, test 1-40

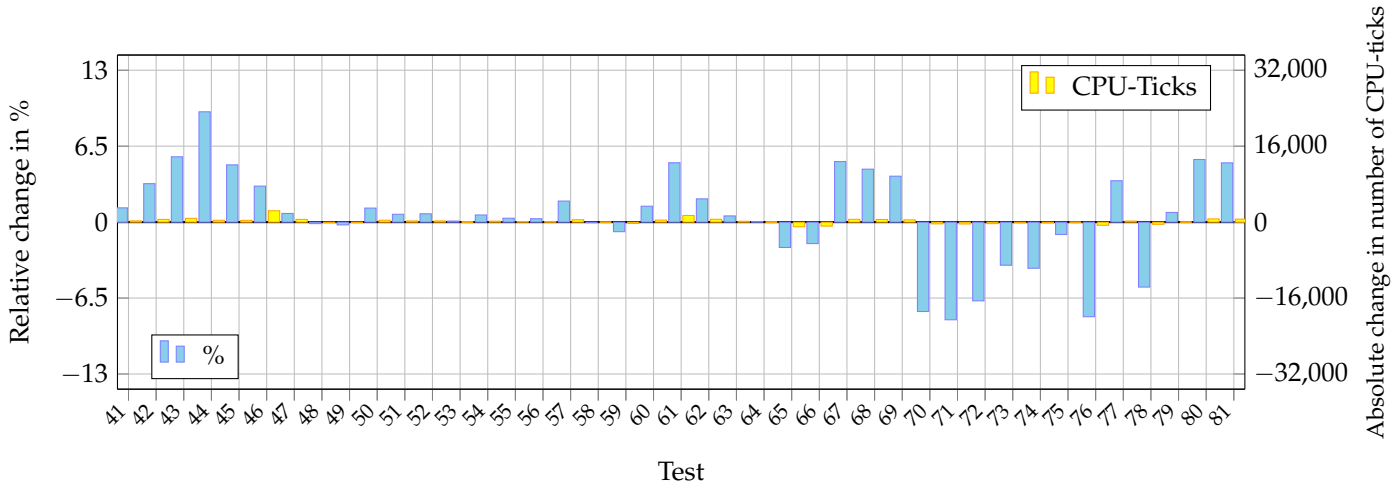


Figure 7.41: Detailed result for Reorder Blocks optimization, test 41-81

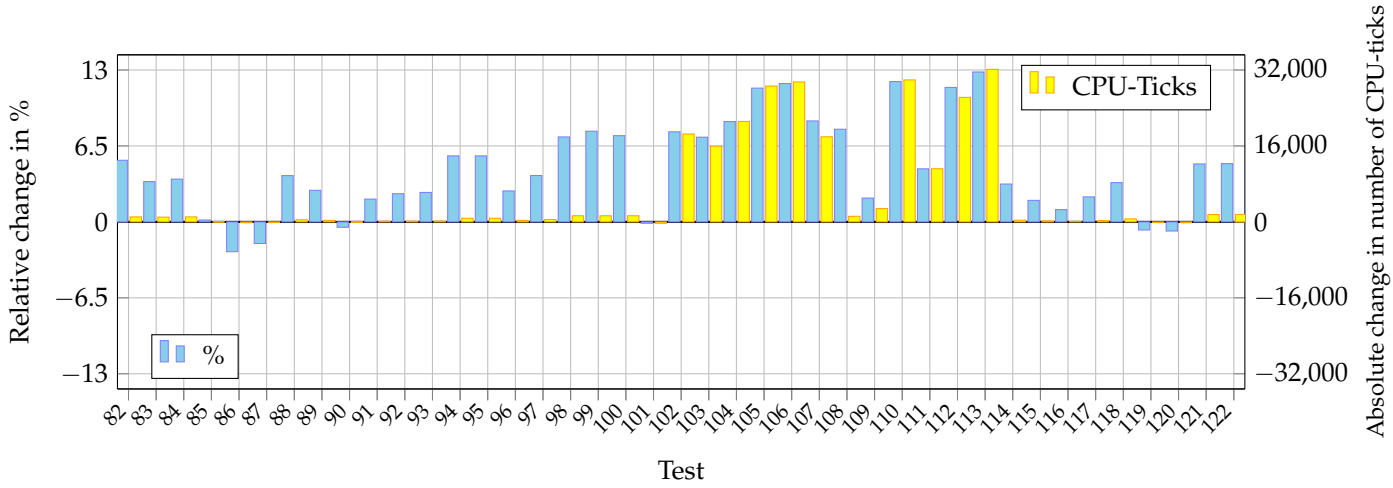


Figure 7.42: Detailed result for Reorder Blocks optimization, test 82-122

7.3.4.3 Expensive optimization

Test	%	Ticks	Test	%	Ticks	Test	%	Ticks	Test	%	Ticks
1	0.22	45	32	-0.42	-28	63	-0.73	-242	93	1.76	108
2	-1.41	-226	33	-0.70	-49	64	-0.69	-214	94	3.18	426
3	-1.02	-168	34	-0.34	-24	65	1.25	516	95	4.08	550
4	2.09	208	35	-0.09	-6	66	1.89	863	96	2.17	257
5	0.63	77	36	-0.06	-4	67	5.81	654	97	3.58	450
6	1.12	107	37	-1.89	-31	68	5.27	644	98	-1.86	-331
7	0.47	47	38	2.80	167	69	4.71	578	99	-1.04	-173
8	0.77	94	39	2.40	185	70	-1.62	-78	100	-1.16	-203
9	0.74	71	40	-5.75	-743	71	-2.72	-128	101	-2.98	-7307
10	0.16	8	41	-1.41	-296	72	0.44	21	102	-2.09	-5016
11	3.37	170	42	-2.37	-406	73	-9.38	-295	103	-1.70	-3742
12	-0.01	-1	43	-0.87	-125	74	-10.20	-320	104	-2.34	-5769
13	-1.53	495	44	5.28	218	75	-5.68	-172	105	-2.42	-6041
14	0.65	132	45	0.11	8	76	0.38	31	106	-1.86	-4640
15	2.06	163	46	-1.14	-887	77	-1.06	-66	107	0.18	364
16	-1.67	133	47	-0.07	532	78	0.87	72	108	3.19	471
17	1.28	101	48	-1.87	-1516	79	0.24	2	109	0.48	664
18	0.00	0	49	-3.28	-1020	80	-1.43	-178	110	-1.65	-4116
19	-0.93	-70	50	-0.20	-66	81	-2.24	-275	111	-3.70	-9133
20	-0.08	-77	51	0.01	5	82	-0.52	-103	112	1.92	4380
21	0.82	16	52	0.08	27	83	-0.62	-179	113	-1.96	-4917
22	0.53	181	53	-2.56	-765	84	-0.07	-21	114	0.09	10
23	3.17	197	54	-0.70	-224	85	4.01	125	115	0.90	98
24	-9.62	-145	55	0.01	3	86	-3.00	-195	116	-0.44	-45
25	-0.07	-13	56	0.03	11	87	-3.22	-209	117	1.43	174
26	6.64	1165	57	-1.04	-291	88	-2.18	-245	118	0.84	159
27	4.69	1005	58	0.07	20	89	-3.61	-413	119	-0.74	-148
28	3.29	693	59	-0.42	-143	90	1.06	157	120	0.80	170
29	0.29	65	60	-0.83	-262	91	1.49	108	121	0.33	103
30	1.21	256	61	0.26	72	92	1.23	71	122	0.20	64
31	2.23	117	62	-0.23	-68						

Table 7.24: Table of detailed values of WCET in percentage for Expensive

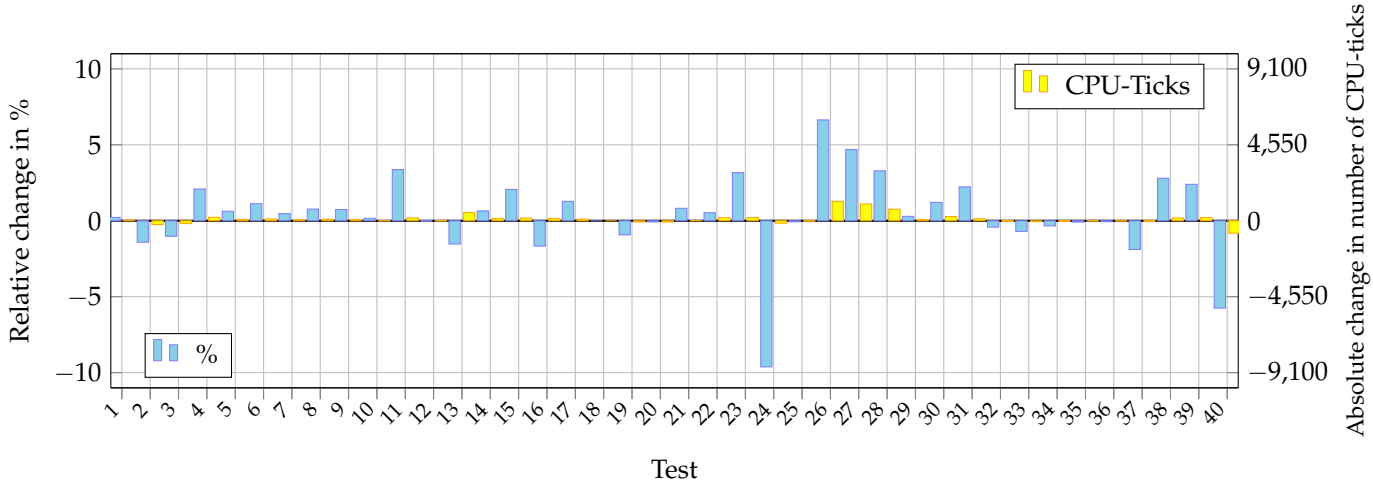


Figure 7.43: Detailed result for Expensive optimization, test 1-40

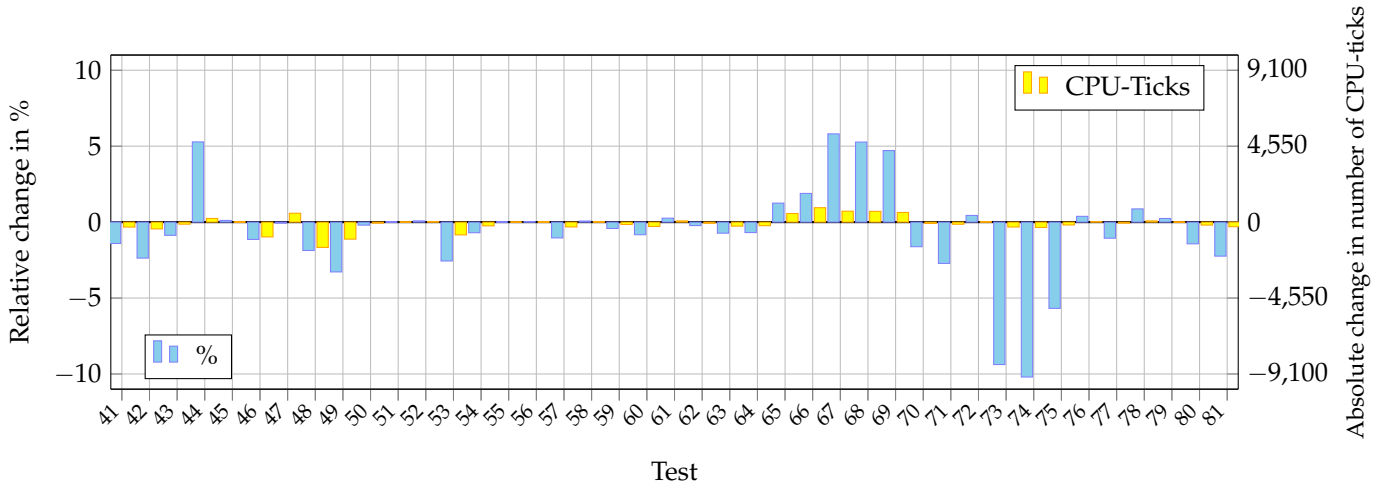


Figure 7.44: Detailed result for Expensive optimization, test 41-81

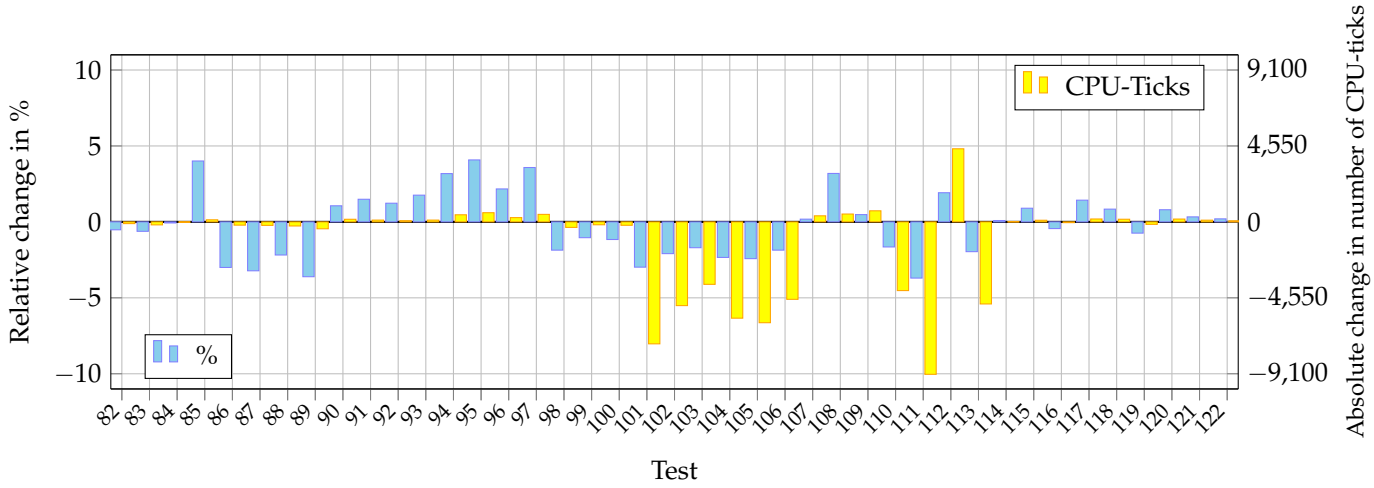


Figure 7.45: Detailed result for Expensive optimization, test 82-122

7.3.5 The largest increase and decrease

In figure 7.25 the largest increase and decrease for the absolute and relative values are shown. The largest increase for both the absolute and relative values are from the same test for the test set Reorder Blocks 7.23. The largest increase was found in the test set for -O3 7.20 for both the absolute and relative values, but the test number differed. The decrease for *test 101* resulted in a absolute change of 16612 CPU-ticks, with the relative value of 6.77%. The decrease for *test 22* with the relative value of -16.01% resulted in an absolute change of 5484 CPU-ticks.

	Absolute			Relative		
	Ticks	Test-set	Test no	%	Test-set	Test no
Increase	32143	Reorder Blocks	113	12.83	Reorder Blocks	113
Decrease	-16612	-O3	101	-16.01	-O3	22

Table 7.25: The highest and lowest values for the absolute and relative unit.