Achieving Software Quality Through Heuristic Transformations: Maintainability and Performance

Bill Andreopoulos

Technical Report CS-2002-05

November 2002

Department of Computer Science and Engineering

4700 Keele Street North York, Ontario M3J 1P3 Canada

Abstract

ACHIEVING SOFTWARE QUALITY THROUGH HEURISTIC TRANSFORMATIONS: MAINTAINABILITY AND PERFORMANCE

Bill Andreopoulos

This report proposes a general framework for evaluating and improving the quality of a software system. To illustrate how the methodology works, the report focuses on the software qualities of maintainability and performance. The Non-Functional Requirements (NFR) framework is adopted to represent and analyse the software qualities of maintainability and performance. Specifically, it analyses the software attributes that affect either quality, the heuristics that can be implemented in source code to achieve either quality, and how the two qualities conflict with each other. Experimental results are discussed to determine the effect of various heuristics on maintainability and performance. A methodology is described for selecting the heuristics that will improve a system's software quality the most.

Contents

1 Introduction **1**

1.1 Objective and Methodology 1

1.1.1 Software Qualities 2

1.1.2 The NFR Framework 3

1.2 Review of Literature 9

1.2.1 Maintainability 9

1.2.2 Performance 11

2 Maintainability and Performance **12**

2.1 Decomposing Maintainability into Softgoals 12

2.2 Decomposing Performance into Softgoals 16

2.3 Identifying Heuristic Transformations to Achieve Software Quality 19

2.3.1 Identifying Heuristics to Satisfice Maintainability 19

2.3.2 Identifying Heuristics to Satisfice Performance 23

3 Maintainability and Performance Measurements **25**

3.1 Maintainability Measurements 26

3.1.1 Maintainability Metrics Models 27

3.1.2 A study of the optimization activities 29

3.1.3 Some conclusions on measuring maintainability 47

3.2 Performance Measurements 48

3.2.1 A study of the optimization activities 49

4 Selecting a Heuristic Transformation **58**

4.1 Validation of the Andre Formula 59

5 Conclusions 61
5.1 Future Work 61

A Glossary 67

B Description of Maintainability Optimization Heuristics 70

C Description of Performance Optimization Heuristics 100

D Maintainability Measurements 123
D.1 Maintainability Metrics Models 123
D.1.1 Documentation Metrics 123
D.1.2 Expression Metrics 124
D.1.3 General Statement Metrics 125
D.1.4 Control-Flow Statement Metrics 125
D.1.5 Executable Statement Metrics 126
D.1.6 Declaration Statement Metrics 126
D.1.7 Nesting Level (Scope) Metrics 127
D.1.8 Cross Reference Metrics 128
D.1.9 McCabe Metric 128
D.1.10 Halstead Metrics 128
D.1.11 Miscellany Metrics 129
D.1.12 Maintainability Indexes 130
D.2 A study of the optimization activities 131
D.2.1 Hoisting and Unswitching 133
D.2.2 Integer Divide Optimization 137
D.2.3 Address Optimization 140
D.2.4 Function Inlining 147
D.2.5 Elimination of GOTO statements 153
D.2.6 Dead Code Elimination 158
D.2.7 Elimination of Global Data Types and Data Structures 162
D.2.8 Maximization of Cohesion 165
D.2.9 Minimization of Coupling Through ADTs 165

List of Tables

3.1 File level maintainability metrics on the WELTAB system before and after hoisting/unswitching . . . 32

3.2 Function level maintainability metrics on the WELTAB system before and after hoisting/unswitching . . . 32

3.3 File level maintainability metrics on the WELTAB system before and after integer divide optimization . . . 33

3.4 Function level maintainability metrics on the WELTAB system before and after integer divide optimization . . . 34

3.5 Function level maintainability metrics on the AVL system before and after integer divide optimization . . . 34

3.6 File level maintainability metrics on the WELTAB system before and after address optimization . . . 35

3.7 Function level maintainability metrics on the WELTAB system before and after address optimization . . . 38

3.8 File level maintainability metrics on the WELTAB system before and after function inlining . . . 39

3.9 Function level maintainability metrics on the WELTAB system before and after function inlining . . . 41

3.10 Function level maintainability metrics on the AVL system before and after function inlining . . . 41

3.11 File level maintainability metrics on the WELTAB system before and after eliminating GOTO statements . . . 42

3.12 Function level maintainability metrics on the WELTAB system before and after eliminating GOTO statements . . . 44

3.13 File level maintainability metrics on the WELTAB system before and after eliminating dead code . . . 45

3.14 Function level maintainability metrics on the WELTAB system before and after eliminating dead code . . . 46
3.15 File level maintainability metrics on the WELTAB system before and after eliminating global data types and data structures . . . 47
3.16 Function level maintainability metrics on the WELTAB system before and after eliminating global data types and data structures . . . 47
3.17 Function level maintainability metrics on the AVL system before and after maximizing cohesion . . . 48
3.18 Function level maintainability metrics on the AVL system before and after minimizing coupling through ADTs . . . 49
3.19 Function level performance metrics on the WELTAB system before and after hoisting and unswitching . . . 50
3.20 Function level performance metrics on the WELTAB system before and after integer divide optimization . . . 51
3.21 Function level performance metrics on the AVL system before and after integer divide optimization . . . 51
3.22 Function level performance metrics on the WELTAB system before and after address optimization . . . 54
3.23 Function level performance metrics on the WELTAB system before and after function inlining . . . 55
3.24 Function level performance metrics on the AVL system before and after function inlining . . . 55
3.25 Function level performance metrics on the WELTAB system before and after elimination of GOTO statements . . . 56
3.26 Function level performance metrics on the WELTAB system before and after dead code elimination . . . 56
3.27 Function level performance metrics on the WELTAB system before and after elimination of global data types and data structures . . . 57
3.28 Function level performance metrics on the AVL system before and after maximizing cohesion . . . 57
3.29 Function level performance metrics on the AVL system before and after minimizing coupling . . . 57

4.1 Testings that show the reliability of Andre formula . . . 60

B.1 Minimization of the depth of the inheritance tree 72
B.2 Minimization of the number of direct children 73
B.3 Minimization of the response set 74
B.4 Maximization of cohesion 75
B.5 Minimization of the summation of McCabe's cyclomatic complexity over all local methods 76
B.6 Dead code elimination 77
B.7 Elimination of GOTO statements 78
B.8 Elimination of global data types and data structures 79
B.9 Initialization integrity 79
B.10 I/O integrity 80
B.11 Minimization of coupling between classes 81
B.12 Minimization of coupling through message passing 82
B.13 Minimization of coupling through inheritance 83
B.14 Minimization of coupling through abstract data types 83
B.15 Maximization of embedded spacing 84
B.16 Good module separation 85
B.17 Good vertical spacing 86
B.18 Good horizontal spacing 87
B.19 Maximization of comment lines within the modules 88
B.20 Comment vague code 89
B.21 Comment each Variable, Type, or Constant Declaration 90
B.22 Appropriate Length of Comments 90
B.23 Maximization of the modules with header (prologue) comments 91
B.24 Include a header comment for each procedure 92
B.25 Include a header comment for each file 93
B.26 Include a header comment for each logical block or module 94
B.27 Good naming conventions 95
B.28 Meaningful names 96
B.29 Reasonable length of names 96
B.30 Good use of symbols and case 97
B.31 Form procedure names with words or abbreviations separated by underscores and use mixed case (e.g., Get_Temp) 98

B.32 Form variable names, class names, and object names with words and abbreviations using mixed case but no underscores (e.g., SensorTemp) . . . 98
B.33 Form names of constants and type definitions using all upper case and using underscores as word separators 99

C.1 Address optimization 102
C.2 Bitfield optimization 103
C.3 Block merging 104
C.4 Branch elimination 105
C.5 Code compression 105
C.6 Code compression in secondary storage 106
C.7 Code compression in main memory 107
C.8 Constant folding 108
C.9 Constant propagation 109
C.10 Common subexpression elimination 109
C.11 Dead code elimination 110
C.12 Elimination of GOTO statements 110
C.13 Expression simplification 111
C.14 Forward store 111
C.15 Function inlining 112
C.16 Hoisting 113
C.17 If optimization 113
C.18 Induction variable elimination 114
C.19 Instruction combining 115
C.20 Integer divide optimization 116
C.21 Integer mod optimization 116
C.22 Integer multiply optimization 117
C.23 Loop collapsing 117
C.24 Loop fusion 118
C.25 Loop unrolling 119
C.26 Narrowing 120
C.27 Peephole optimization 120
C.28 Tail recursion 121
C.29 Unswitching 121

C.30 Value range optimization 122

D.1 File level maintainability metrics on the WELTAB system before and after Hoisting/Unswitching 136
D.2 Function level maintainability metrics on the WELTAB system before and after hoisting/unswitching 137
D.3 File level maintainability metrics on the WELTAB system before and after integer divide optimization 139
D.4 Function level maintainability metrics on the WELTAB system before and after integer divide optimization 140
D.5 Function level maintainability metrics on the AVL system before and after integer divide optimization 140
D.6 File level maintainability metrics on the WELTAB system before and after address optimization 143
D.7 Function level maintainability metrics on the WELTAB system before and after address optimization 147
D.8 File level maintainability metrics on the WELTAB system before and after Function Inlining 150
D.9 Function level maintainability metrics on the WELTAB system before and after function inlining 152
D.10 Function level maintainability metrics on the AVL system before and after function inlining 153
D.11 File level maintainability metrics on the WELTAB system before and after eliminating GOTO statements 155
D.12 Function level maintainability metrics on the WELTAB system before and after eliminating GOTO statements 158
D.13 File level maintainability metrics on the WELTAB system before and after eliminating dead code 160
D.14 Function level maintainability metrics on the WELTAB system before and after eliminating dead code 162
D.15 File level maintainability metrics on the WELTAB system before and after eliminating global data types and data structures 164
D.16 Function level maintainability metrics on the WELTAB system before and after eliminating global data types and data structures 165

D.17 Function level maintainability metrics on the AVL system before and after maximizing cohesion 165
D.18 Function level maintainability metrics on the AVL system before and after minimizing coupling through ADTs 166

List of Figures

1.1 Example of a softgoal interdependency graph 7
1.2 Selecting among alternative combinations of heuristics 8

2.1 Maintainability softgoal interdependency graph 13
2.2 Performance softgoal interdependency graph 21
2.3 Maintainability softgoal interdependency graph, including heuristics . . . 22
2.4 Performance softgoal interdependency graph, including heuristics 24

B.1 Maintainability softgoal interdependency graph, including heuristics . . . 71

C.1 Performance softgoal interdependency graph, including heuristics 101

Chapter 1

Introduction

Software quality has become a topic of major concern in the field of software engineering. As organizations rely increasingly upon software to perform their vital functions, building software of high quality becomes a critical issue. The technical quality of software may seriously affect various organizational activities, such as the delivery of services, the administration, or even the amount of software maintenance required. The cost of organizational operations can increase substantially as software quality decreases. Thus, it is necessary to be able to represent and analyze software quality effectively throughout the entire software life-cycle.

1.1 Objective and Methodology

The main goal of this report is to propose a general framework for evaluating and improving the quality of a software system. To illustrate how the methodology works, we focus in this report on the *maintainability* and *performance* software qualities, since software development experience has shown that these are two extremely important quality requirements for any software system. Furthermore, the research that has been put into understanding these qualities has failed to eliminate controversy over how to achieve them. More specifically, we examined:

1. the software attributes (or characteristics) that affect one or both qualities,
2. the heuristic transformations (or heuristics) that can be implemented in a software system at the source code level to achieve one or both desired qualities, and
3. how the two qualities conflict with each other.

Section 1.2.1 discusses software qualities in general, and section 1.2.2 describes how qualities can be represented and analysed using the NFR framework. [1]

Most of the information presented in this report was gathered from various research experiments on software quality, which focused on a single software attribute or heuristic. A thorough description of the sources consulted in this report is given in Section 1.3.

Chapter 2 adopts the NFR Framework to represent the qualities of maintainability and performance and their inter-dependencies. The results of our research were also encoded in XML files, and made available on the World Wide Web (WWW) for use by software developers. The URL is:

`http://www.cs.yorku.ca/~billa/SIG/SIG.xml`

The purpose of this URL is to provide a tool that can be used by software developers for optimizing a software system at the source code level. This URL can be used to select the set of heuristics that will benefit the system's maintainability and/or performance the most, while minimizing the negative side-effects.

In many cases, the relevant literature contained gaps in explaining how maintainability and performance conflict with each other, or how they are affected by different heuristics. In such cases we conducted experiments ourselves, to justify our claims on the basis of empirical data. A thorough description of our experiments is given in Chapter 3.

Chapter 4 presents how this methodology can be used to select the heuristic transformations that will improve the system's software quality the most.

Finally, the *Glossary (Appendix A)* gives precise definitions for most of the terms mentioned throughout this report.

1.1.1 Software Qualities

In requirements engineering, a requirement can be described as a condition or capability to which a system must conform, and which is either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document. Requirements can be classified into:

- Functional requirements, which are externally visible behaviors, showing what the system must do, and

- Non-functional requirements (or *software qualities*), which are constraints on the design and/or implementation of the solution.

Software qualities describe not *what* the software will do, but *how* the software will do it, by specifying constraints on the system design and/or implementation. Some types of qualities are:

Process requirements: development requirements, delivery requirements, organization standards (e.g. Use VB v6.0, conform to D0-178B).

Product requirements: usability, capacity, reliability, availability, maintainability, portability, etc.

Real-time constraints: both periodicity and response times. Under some circumstances, these might be considered to be functional requirements.

External requirements: legislative requirements cost constraints, inter-operability.

Unfortunately, software qualities are usually specified briefly and vaguely for a particular system, since no exact techniques for representing them have been standardized yet by the software engineering community. When it comes to modelling qualities, it is common to simply use natural language.

1.1.2 The NFR Framework

The *NFR framework* for representing software qualities was developed by Lawrence Chung, Brian Nixon, Eric Yu and John Mylopoulos at the University of Toronto. [1] The NFR framework represents quality requirements as *softgoals*. Softgoals are goals with no clear-cut criterion for their fulfilment. Instead, a softgoal may only contribute positively or negatively towards achieving another softgoal. By using this logic, a softgoal can be *satisficed* or not. In the NFR framework, *satisficing* refers to satisfying at some level a goal or a need, but without necessarily producing the optimal solution. [1]

The NFR framework represents information about softgoals using primarily a graphical representation, called the *softgoal interdependency graph*. A softgoal interdependency graph records all softgoals being considered, as well as the interdependencies between them. [1] An example of a softgoal interdependency graph is given in Figure 1.1.

In a *softgoal interdependency graph* each softgoal is represented as an individual node (or cloud). A developer can construct an initial *softgoal interdependency graph* by identifying the top-level quality requirement that the system is expected to meet and sketching a softgoal for it. Figure 1.1 shows the maintainability quality requirement as a softgoal at the top of the graph. The softgoal interdependency graph provides a hierarchical arrangement of all the different softgoals; more general *parent softgoals* are shown above more specific *offspring softgoals*. In Figure 1.1 the general *high maintainability* softgoal gets decomposed into the more specific *high source code quality* and *high documentation quality* softgoals.

Softgoals are connected by interdependency links, which show *decompositions* of parent softgoals downwards into more specific offspring softgoals. In some cases the interdendency links are grouped together with an arc; this is referred to as an *AND* contribution of the offspring softgoals towards their parent softgoal, and means that both offspring softgoals must be satisficed to satisfice the parent. Figure 1.1 shows that both softgoals for *high source code quality* and *high documentation quality* must be satisficed to satisfice the *high maintainability* softgoal. In other cases the interdendency links are grouped together with a double arc; this is referred to as an *OR* contribution of the offspring softgoals towards their parent softgoal, and means that only one offspring softgoals needs to be satisficed to satisfice the parent. Figure 1.1 shows that either *low span of data* or *high data consistency* needs to be satisficed to satisfice the *high information structure quality* softgoal.

The bottom of the graph consists of the heuristic transformations (or *heuristics*) that can be directly implemented in the system, to achieve one or more parent softgoals. Figure 1.1 illustrates the *dead code elimination*, *minimization of the response set* and *minimization of the number of direct children* heuristics. Like other softgoals, heuristics (or operationalizing softgoals) also make a contribution towards one or more parent softgoals. Interdependency links show these contributions. In this case, a heuristic's contribution towards satisficing a parent softgoal can be positive ("+" or "++") or negative ("-" or "–"). [1]

The softgoal interdependency graph is incrementally constructed, analysed and revisioned at each step of development (e.g. requirements specification, implementation, etc), to record the developer's consideration of softgoals at that step. Thus, at each step of development the developer can look at information concerning softgoals relevant only to that step of the process. [1]

Chapter 2 illustrates how the NFR framework can be used, to create a detailed decomposition of the maintainability and performance qualities into softgoals.

Selecting the Heuristic Transformations to be Implemented in the Target System

When choosing a set of heuristic transformations (or *heuristics*) to be implemented in the target system, an *evaluation procedure* can be used to determine the degree to which each top-level quality requirement (i.e. maintainability) will be achieved.

In the NFR framework achieving a quality requirement is thought to be a matter of degree, not a binary true-false condition. The set of heuristics selected must be the one which will benefit the system the most, by maximizing the ratio of gains to losses. Thus when evaluating alternative sets of heuristics, one has to consider all gains and losses for each set.

Our goal is to briefly illustrate the evaluation procedure which the NFR framework provides for selecting among alternatives.

In the softgoal interdependency graph, the heuristics that are chosen to be implemented (or satisficed) in the target system are indicated by "$\surd$". On the other hand, rejected candidates are represented as "X". Heuristics for which a decision has not been made are simply left blank.

Suppose the developer selects the *dead code elimination* heuristic, for satisficing *high control flow consistency* and *high data consistency*. Suppose the developer also selects the *minimization of the number of direct children* and *minimization of the response set* heuristics to satisfice *low control flow complexity*. All these selections are represented in Figure 1.2 as check-marks ("$\surd$") inside the nodes.

After the developer selects the heuristics to be implemented, he/she has to evaluate the precise impact of these selections on top-level quality requirements (i.e. maintainability). This will indicate whether the top-level quality requirements are achieved or not.

The evaluation process can be viewed as working bottom-up, starting with bottom leaves of the graph representing heuristics. The evaluation process works towards the top of the graph, determining the impact of offspring softgoals on parent softgoals. This impact is represented by assigning labels ("$\surd$" and "X") to the higher-level parent softgoals.

The impact upon a parent softgoal is computed from the contributions that all the offspring softgoals make towards it. Roughly speaking, when there is a single offspring, a positive contribution "propagates" the offspring's label to the parent. Thus a satisficed offspring results in a satisficed parent, and a denied offspring results in a denied parent. On the other hand, a negative contribution will take the offspring's label and "invert" it for the parent's label. Thus a satisficed offspring results in a denied parent, and a denied offspring results in a satisficed parent.

This is shown in Figure 1.2. The heuristic *minimization of the number of direct children* which is satisficed ("$\surd$"), makes a negative contribution towards its parent softgoal *high module reuse*. The result is that softgoal *high module reuse* is denied ("X"). On the other hand, the heuristic *dead code elimination* which is satisficed ("$\surd$"), makes a positive contribution towards its parent softgoals *high control flow consistency* and *high data consistency*. Thus, both softgoals are satisficed ("$\surd$").

Suppose a softgoal receives contributions from more than one offspring. Then the contribution of each offspring towards the parent is determined, using the above approach, and the individual results are then combined. For example, *low control flow complexity* has two offsprings that are satisficed, and both make a positive contribution towards satisficing the parent. Thus, the combination of their individual positive results is to satisfice the parent softgoal *low control flow complexity*. This is shown in Figure 1.2.

In cases where there is a combination of positive and negative contributions towards a parent softgoal, it is hard to assign a precise value to it. The parent softgoal could be satisficed, denied, or something in between, *depending* on the specific situation. In these cases, a designer can decide whether the parent softgoal is satisficed or not, by considering the rationale recorded as underlying the positive and negative contributions to the parent.

To complete this example, we need to show how all these contributions propagate upwards towards the top-level quality requirements. *High control flow consistency*, *low control flow complexity* and *high module reuse* participate in an OR contribution towards their parent, *high control structure quality*. Since at least one of the offspring softgoals is satisficed, *high control structure quality* is automatically evaluated to be satisficed ("$\surd$").

In this example we have shown how a set of heuristics would contribute towards the maintainability quality only. In order to assess how well the target system would meet all qualities of interest, it would also be necessary to consider the contributions of the selected heuristics towards the performance quality.

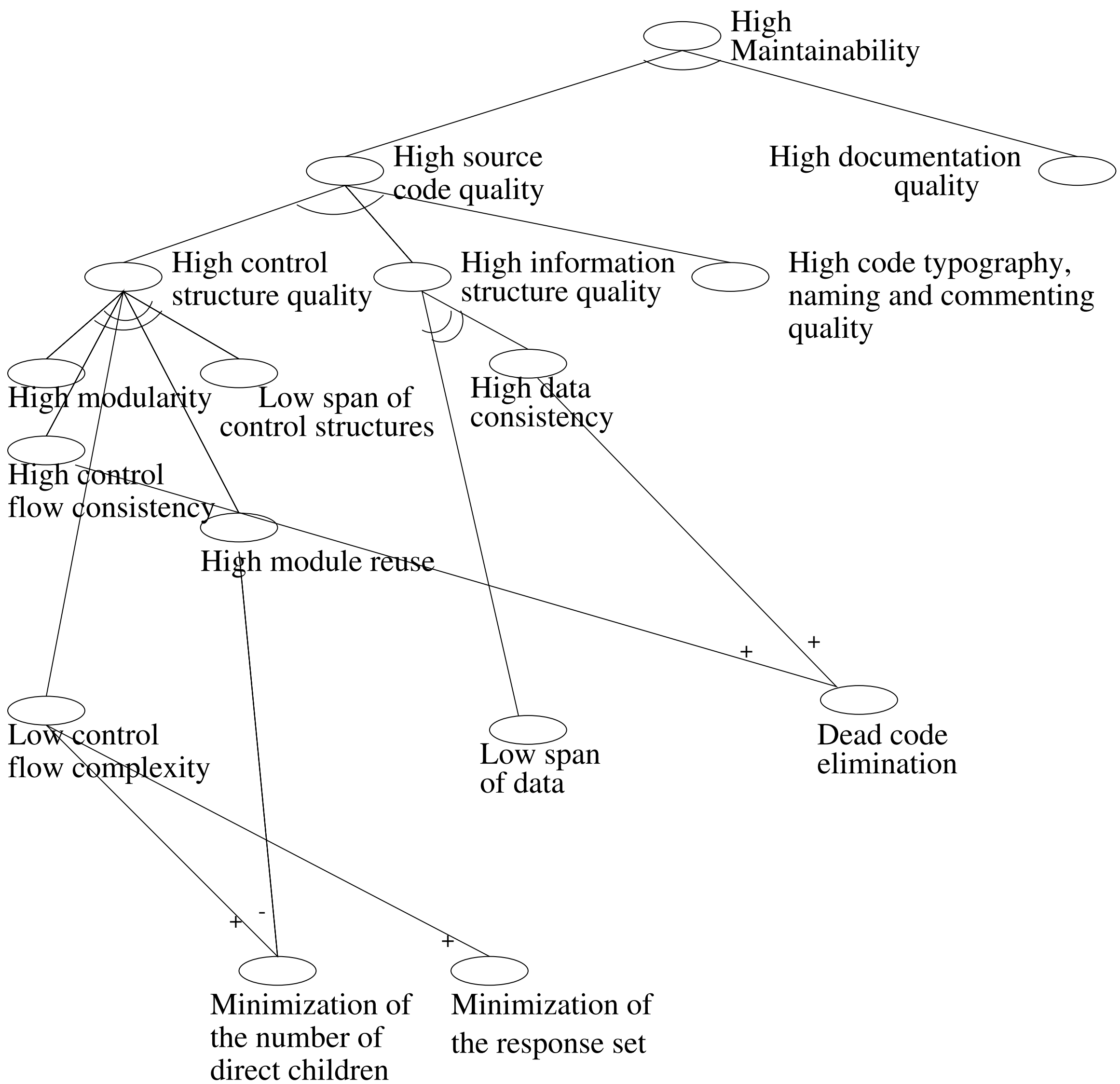

Figure 1.1: Example of a softgoal interdependency graph

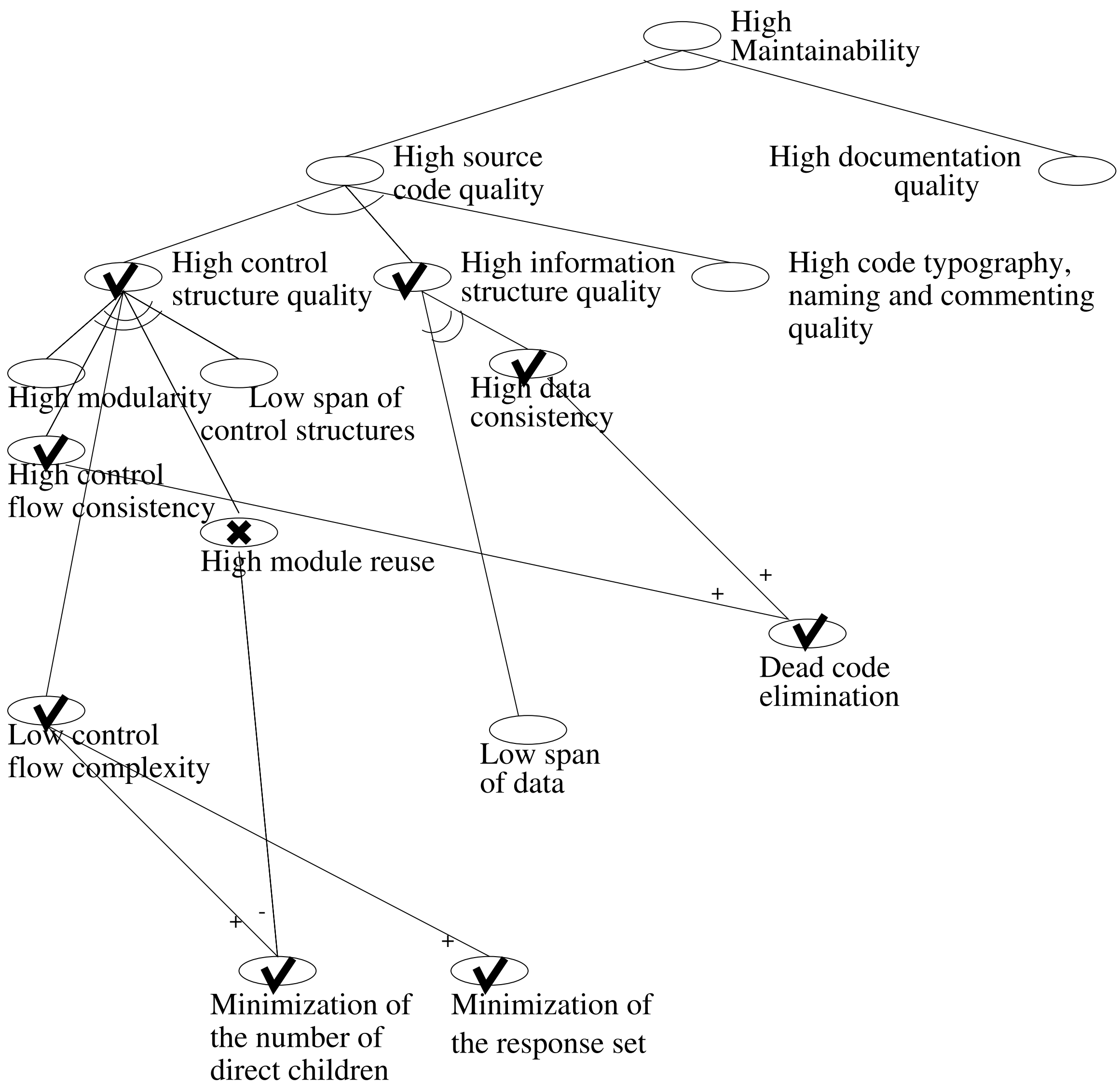

Figure 1.2: Selecting among alternative combinations of heuristics

1.2 Review of Literature

This Section gives an overview of the literature that provided us with information on the software qualities of maintainability and performance. We tried to ensure that this literature would cover relevant past work as extensively as possible.

It is important to note that the maintainability-related literature was much broader than the performance-related literature. Two reasons can be identified to justify this discrepancy:

1. It is much more difficult to measure maintainability precisely than it is to measure performance. Performance metrics have been accepted and used with confidence. However, researchers have failed to agree on a set of metrics to measure maintainability effectively.

2. It is difficult to identify heuristics that can be implemented in source code at a low-level to improve maintainability. Maintainability itself is a qualitative concept; many experimental studies are required before one can argue with confidence about the effects of a heuristic upon software maintainability.

The definition of terms given throughout the report were adopted from [2, 3, 4, 5].

1.2.1 Maintainability

The most comprehensive source of ideas for decomposing *maintainability* into softgoals was the Master's thesis "A Metric Approach to Assessing the Maintainability of Software", written by Jack Hagemeister at the University of Idaho in 1992 [6]. In this work, a hierarchical tree structure of software attributes that affect maintainability is defined. This hierarchical tree structure is refined through successive subtrees until a leaf node representing a low-level software attribute is identified and defined. Hagemeister analysed many published works on software maintainability to define this hierarchical tree structure.

The main source of information on the effects of inheritance on maintainability was the paper "A Study on the Effect of Architecture on Maintainability of Object-Oriented Systems" by P. Hsia, A. Gupta, C. Kung, J. Peng and S. Liu [7]. This paper presents a study indicating that the structure of the inheritance hierarchy of an object-oriented system affects its maintainability. Another similar paper is "The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study" by J. Daly, A. Brooks, J. Miller, M. Roper and M. Wood [8]. This paper presents a series of experiments to show the effect of inheritance on the maintainability of object-oriented software.

The main source of information on the effects of modularity on maintainability was the paper "An Experiment of Legacy Code Segmentation to Improve Maintainability" by R. Panteado, P. Masiero and M. Cagnin [9]. This paper reports an experiment whose purpose is to segment procedural code into modules, to improve system maintainability.

The main source of information on the effects of encapsulation on maintainability was the paper "A Modified Inheritance Mechanism Enhancing Reusability and Maintainability in Object-Oriented Languages" by L. XuanDong and Z. GuoLiang. This paper describes encapsulation problems that may result from use of inheritance. It also presents a modified inheritance mechanism which overcomes these encapsulation problems.

The main source of information on the effects of coupling and cohesion on maintainability was the paper "Measuring and Assessing Maintainability at the End of High Level Design" by L. Briand, S. Morasca and V. Basili [10]. This paper presents a measurement approach for cohesion and coupling, based on object-oriented design principles. A similar paper is "System Architecture Metrics for Controlling Software Maintainability" by M.J. Shepperd. This paper reports the results of an investigation into the relationship between information flow based metrics and software maintainability. It shows that there exists a strong correlation between module maintainability and module information flow.

It was necessary to examine work on software maintainability metrics models. Numerous papers were found on this subject. The most important paper for our work was "Constructing and Testing Software Maintainability Assessment Models" by F. Zhuo, B. Lowther, P. Oman and J. Hagemeister [11]. This paper presents and compares seven software maintainability assessment models. These models are mostly based on Halstead's effort, extended cyclomatic complexity, lines of code, and number of comments. A similar paper is "Using Software Maintainability Models to Track Code Health" by D. Ash, J. Alderete, L. Yao, P. Oman and B. Lowther [12]. This paper also describes mechanisms for software maintainability assessment.

Finally, it was necessary to examine related work that has been done in the area of software reengineering. A good paper on this subject was "A study on the Effect of Reengineering upon Software Maintainability" by H. Sneed and A. Kaposi [13]. This paper examines how restructuring and reengineering can be applied to software to improve maintainability. It shows that restructuring the program (e.g. by eliminating GOTO statements) reduces the maintenance effort. Another similar paper is "Effect of Object Orientation on Maintainability of Software" by G. Aditya Kiran, S. Haripriya and P. Jalote [14]. This paper describes an experimental study about the effect of object orientation on maintenance. It shows that object oriented software generally has better maintainability.

1.2.2 Performance

Many ideas for developing the *performance* decomposition were taken from the textbook "Computer Architecture: A Quantitative Approach" written by David Patterson and John Hennessy. [15] This book explains that performance can be defined in terms of speed (time performance) or in terms of storage requirements (space performance), depending on our purposes. [15]

Furthermore, most of the performance optimization heuristics were provided by the Ph.D. thesis "Fast and Effective Optimization of Statically Typed Object-Oriented Languages", written by D.F.Bacon at the University of California, Berkeley, in 1997. [16]. Bacon's Ph.D. thesis was found to be the most comprehensive source of information on this subject.

Finally, Brian Nixon's work on performance requirements [17, 18] has many similarities to our work and contributed many ideas to our research. Nixon applied the NFR framework to represent and organize performance requirements. The result of his work was a specialization of the NFR framework, the *Performance Requirements Framework*. This framework represents the basic performance softgoals, such as time and space, and provides a notation for describing performance requirements. [17, 18]

Chapter 2

Maintainability and Performance

This chapter can be viewed as an analysis of the *maintainability* and *performance* qualities for a system, followed by a synthesis of heuristic transformations (or heuristics) to improve these qualities. Specifically, we use the NFR framework presented in Chapter 1 to examine in detail the maintainability and performance qualities.

Sections 2.1 and 2.2 describe the *softgoal interdependency graphs* built for maintainability and performance respectively, by systematically decomposing the general qualities into specific softgoals. Section 2.3 explains how the qualities of maintainability and performance can be satisficed in a system, by implementing specific heuristics at a low-level. The *Glossary (Appendix A)* gives precise definitions for most of the terms mentioned in this section.

2.1 Decomposing Maintainability into Softgoals

Maintainability is defined as the characteristics of the software, its history, and associated environments that affect the maintenance process and are indicative of the amount of effort necessary to perform maintenance changes. It can be measured as a quantification of the time necessary to make maintenance changes to the product. [3, 6]

The initial maintainability quality is quite broad and abstract. Researchers have determined numerous and varied attributes of software which might affect maintainability. To effectively deal with such a broad quality, we treat it as a *softgoal* (see Section 1.2) and then decompose it down into more specific softgoals.

It is important to note that in this work we only describe softgoals relevant to the source code of the target system. It is possible to identify softgoals irrelevant to source code, that contribute towards satisficing maintainability. Such softgoals may be related to other environmental factors, such as 'Management' or the 'Operational Environment'. [6] However, identifying such heuristics would require knowledge about the specific environment in which the software system is embedded, and thus describing them is outside the scope of our work.

Figure 2.1 shows the full *softgoal interdependency graph* for maintainability. This graph attempts to illustrate the specific software attributes that affect maintainability. In cases where there exist conflicting views of how attributes affect the maintainability of software, these cases are noted throughout our descriptions.

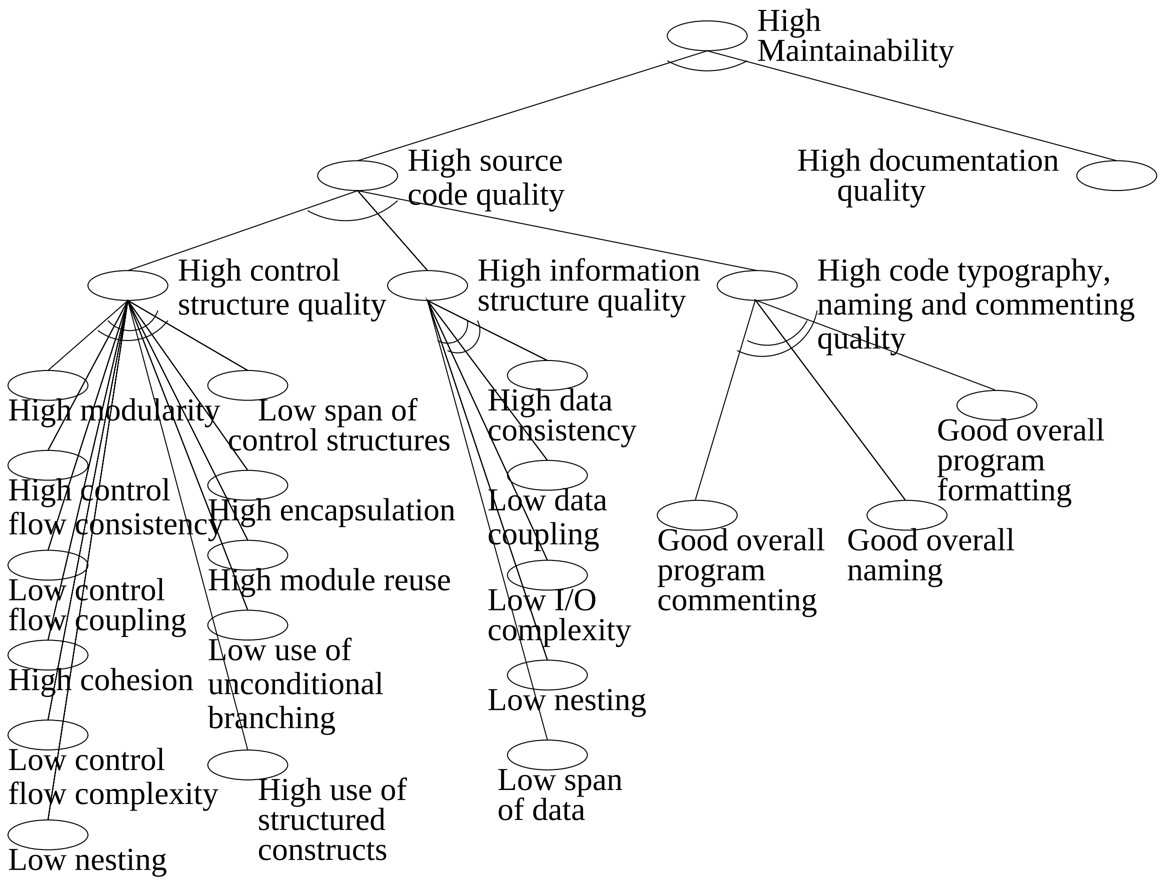

Figure 2.1: Maintainability softgoal interdependency graph

The maintainability quality can be decomposed into softgoals

- high source code quality [6], and
- high documentation quality [19].

This decomposition is shown in Figure 2.1.

Both softgoals of high source code and documentation quality must be satisficed for a system to have high maintainability. This is referred to as an *AND* contribution of the offspring softgoals towards their parent softgoal, and is shown by grouping the interdependency lines with an arc. The rationale behind this *AND* contribution is that a software system with clear source code but bad documentation will be hard to maintain, since maintainers will need to study requirements and design documents in order to understand how the system works. A software system with clear documentation but badly-written code will also be hard to maintain, since maintainers will need to understand how the source code works in order to make changes to it. Thus, software developers must try to satisfice both softgoals in a system.

The *high source code quality* softgoal can be further decomposed into the sub-softgoals

- high control structure quality [6],
- high information structure quality [6], and
- high code typography, naming and commenting quality [20, 21].

This decomposition is shown in Figure 2.1. As shown, this is also an *AND* contribution, i.e. all three sub-softgoals must be satisficed to achieve the *high source code quality* softgoal. The rationale behind this *AND* contribution is that source code will be hard to understand if it is badly commented, or is laid out in a bad manner (typography qualities). But source code will also be hard to understand if characteristics such as modularity, encapsulation or cohesion have not been achieved (control structure and information structure qualities).

Now we want to focus on each of these sub-softgoals individually. The *high control structure quality* softgoal can further be decomposed into the sub-softgoals that source code must be characterized by the following attributes:

- high modularity [22, 23, 24, 25] ,
- high control flow consistency [6],

- low control flow coupling [26, 10, 27] ,
- high cohesion [26, 10, 27],
- low control flow complexity [25],
- low nesting [6],
- low span of control structures [28, 29],
- high encapsulation [30],
- high module reuse [6],
- low use of unconditional branching [6],
- high use of structured constructs [28, 29].

This decomposition is shown in Figure 2.1. As shown, this is an *OR* contribution, i.e. it is *not* necessary for *all* of the sub-softgoals to be satisficed to achieve the *high control structure quality* softgoal. This is shown with the interdependency lines grouped by a double arc. The rationale behind this *OR* contribution is that the softgoals which affect a system's control structure often overlap with each other, and satisficing all of them simultaneously may be impossible to achieve. For example, by satisficing *Low use of unconditional branching* one may affect negatively *Low control flow complexity.* Thus, it would not make sense to claim that all softgoals which affect the control structure need to be satisficed. Instead, by satisficing some of these softgoals a developer can feel confident that the system is characterized by *high control structure quality.*

The *high information structure quality* softgoal can further be decomposed into the sub-softgoals that source code must be characterized by the following attributes:

- high data consistency [6, 28, 29],
- low data coupling [6, 28, 29],
- low I/O complexity [6, 28, 29],
- low nesting [6, 28, 29],
- low span of data [6, 28, 29].

This decomposition is shown in Figure 2.1. As shown, this is also an *OR* contribution, i.e. it is *not* necessary for *all* of the sub-softgoals to be satisficed to achieve the *high information structure quality* softgoal. The rationale behind this *OR* contribution is that the softgoals which affect a system's information structure often overlap with each other, and satisficing all of them simultaneously may be impossible to achieve. Thus, it would not make sense to claim that all softgoals which affect the information structure need to be satisficed. Instead, by satisficing a reasonable number of these softgoals a developer can feel confident that the system is characterized by *high information structure quality.*

The *high code typography, naming and commenting quality* softgoal can further be decomposed into the sub-softgoals that source code must be characterized by the following attributes:

- good overall program formatting [21, 20],
- good overall program commenting [31, 21, 20],
- good overall naming [21, 20].

This decomposition is shown in Figure 2.1. As shown, this is also an *OR* contribution, i.e. it is *not* necessary for *all* of the sub-softgoals to be satisficed to achieve the *high code typography, naming and commenting quality* softgoal. The rationale behind this contribution is that the softgoals which affect a system's typography often overlap with each other, and satisficing all of them simultaneously may be impossible to achieve. Thus, it would not make sense to claim that all softgoals which affect typography need to be satisficed. Instead, by satisficing a reasonable number of these softgoals a developer can feel confident that the system is characterized by *high code typography, naming and commenting quality* .

2.2 Decomposing Performance into Softgoals

As with maintainability, we also view performance as a *softgoal* (see Section 1.2) that can be broken down into more specific softgoals. Figure 2.2 shows the full *softgoal interdependency graph* for performance.

The *high performance* quality can be decomposed into softgoals

- good time performance [15], and

- good space performance [15].

This decomposition is shown in Figure 2.2. As shown, this is an *AND* contribution, i.e. both softgoals must be satisficed to achieve the *performance* softgoal. The rationale behind this *AND* contribution is that both softgoals of good time and space performance must be satisficed for a system to achieve good performance. It is inconceivable for a system which is fast but makes bad memory-utilization to be characterized by good performance. It is also inconceivable for a system which makes good memory-utilization but is slow to be characterized by good performance. Thus, software developers must try to satisfice both softgoals in a system. If there is a tradeoff involved between achieving both of them then that tradeoff must be balanced.

In turn, the *good space performance* softgoal can be decomposed into the following sub-softgoals:

- low main memory utilization, and

- low secondary storage utilization.

This decomposition is shown in Figure 2.2. As shown, this is also an *AND* contribution, i.e. both sub-softgoals must be satisficed to achieve the *good space performance* softgoal. The rationale behind this *AND* contribution is that the system may be stored either in main memory or in secondary storage, and the term "space" is used interchangeably to refer to both types of storage.

The *good time performance* softgoal can be decomposed into the following sub-softgoals:

- low response time, and

- high throughout.

This decomposition is shown in Figure 2.2. As shown, this is an *OR* contribution. The rationale behind this *OR* contribution is that in most cases a developer will focus on either response time or throughput in an attempt to improve time performance. Throughput and response time are related to each other, because decreasing response time almost always improves throughput. Furthermore, the goal of achieving low response time or high throughput usually depends on the specific situation being considered. For example, if a program is running on two different workstations, then the faster workstation would be the one that gets the job done first, i.e. the one with the lowest response time.

However, if jobs were submitted by many users to each of these workstations, then the faster workstation would be the one that completed the most jobs during a day, i.e. the one with the highest throughput. [15]

In turn, the *low response time* softgoal can be decomposed into the following sub-softgoals:

- low CPU time,
- low I/O activities, and
- low time running other programs.

This decomposition is shown in Figure 2.2. As shown, this is an *OR* contribution. The rationale behind this *OR* contribution is that a program's response time can be improved by decreasing either the time spent running other programs, or time spent for I/O activities, or the CPU time. Thus, it is not necessary to achieve all of the sub-softgoals in order to achieve low response time.

The *low CPU time* softgoal can be decomposed into the following sub-softgoals:

- low user CPU time, and
- low system CPU time.

This decomposition is shown in Figure 2.2. As shown, this is also an *OR* contribution. The rationale behind this *OR* contribution is that a program's CPU time can be improved by decreasing either the user CPU time or the system CPU time. Thus, it is not necessary to achieve all of the sub-softgoals in order to achieve low CPU time. Furthermore, the distinction between user CPU time and system CPU time is often blurry, and in such cases it might not make sense to speak of achieving both softgoals.

The *low system CPU time* softgoal can be decomposed into the following sub-softgoals:

- low disk access, and
- low memory access.

This decomposition is shown in Figure 2.2. As shown, this is also an *OR* contribution. The rationale behind this *OR* contribution is that a program's system CPU time can be improved by decreasing either disk accesses or memory accesses. Thus, it is not necessary to achieve all of the sub-softgoals in order to achieve low system CPU time.

2.3 Identifying Heuristic Transformations to Achieve Software Quality

Up to now we have been providing more precise definitions for the broad qualities of maintainability and performance. However, we have not yet described the means by which one could achieve high maintainability and performance in a system.

At this point we have reached our original destination, which is to identify the heuristic transformations (or *heuristics*) that actually satisfice the quality requirements of high maintainability and performance, and then to select the best combination of heuristics for the target system. In Section 1.2.2 we showed how the NFR framework could be used to select the best combination of heuristics.

The NFR framework treats these heuristics as *softgoals* (see Section 1.2), because this allows developers to decompose heuristics into more specific ones. Heuristics are often referred to as *operationalizing softgoals*.

Like other softgoals, heuristics also make a contribution towards one or more parent softgoals. In this case the contribution types are positive/negative. This is represented with a " + ", " + +", or " − ", " − −" symbol. [1]

2.3.1 Identifying Heuristics to Satisfice Maintainability

In this section we briefly describe some of the *heuristics* that can be implemented in a system's source code to contribute towards satisficing the maintainability quality requirement. Appendix B provides a full description of all the maintainability heuristics as well as their contributions, and should be consulted for further details.

The softgoal interdependency graph given in Figure 2.3 illustrates a subset of these heuristics as well as their contributions towards their parent softgoals. A more complete version of this graph illustrating the entire set of heuristics can be found in Figure B.1.

As shown in Figure 2.3, an example of a maintainability heuristic is *dead code elimination*. This means to eliminate code that is unreachable or that does not affect the program (e.g. dead stores). Implementing this heuristic makes a "++" contribution towards meeting the *high control flow consistency* and *high data consistency* softgoals. Dead code elimination may also affect performance in various ways. We discuss these contributions in the next section.

As shown in Figure 2.3, another example of a maintainability heuristic is *elimination* *of GOTO statements*. This means to minimize the number of GOTO statements in the source code. Implementing this heuristic makes a "++" contribution towards meeting the *low use of unconditional branching* softgoal. Implementing this heuristic also makes a "-" contribution towards meeting the *low control flow complexity* softgoal. Elimination of GOTO statements may also affect performance in various ways. We discuss these contributions in the next section.

As shown in Figure 2.3, another example of a maintainability heuristic is *elimination of global data types and data structures*. This means to make global data types and data structures local. Implementing this heuristic makes a "++" contribution towards meeting the *low data coupling* softgoal.

A full discussion of the rest of the maintainability heuristics and their contributions towards their parent softgoals can be found in Appendix B.

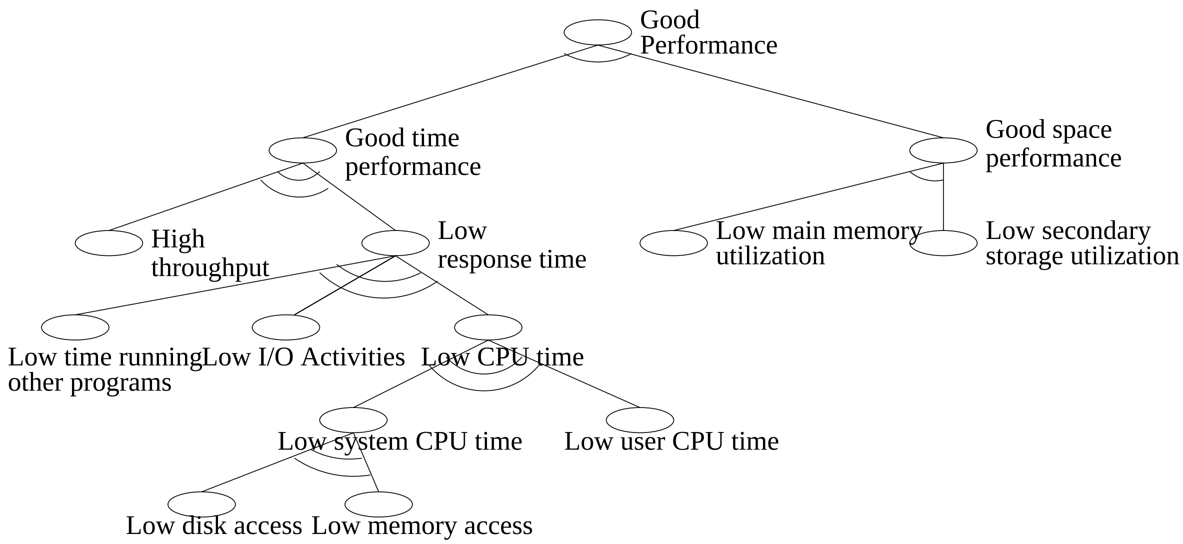

Figure 2.2: Performance softgoal interdependency graph

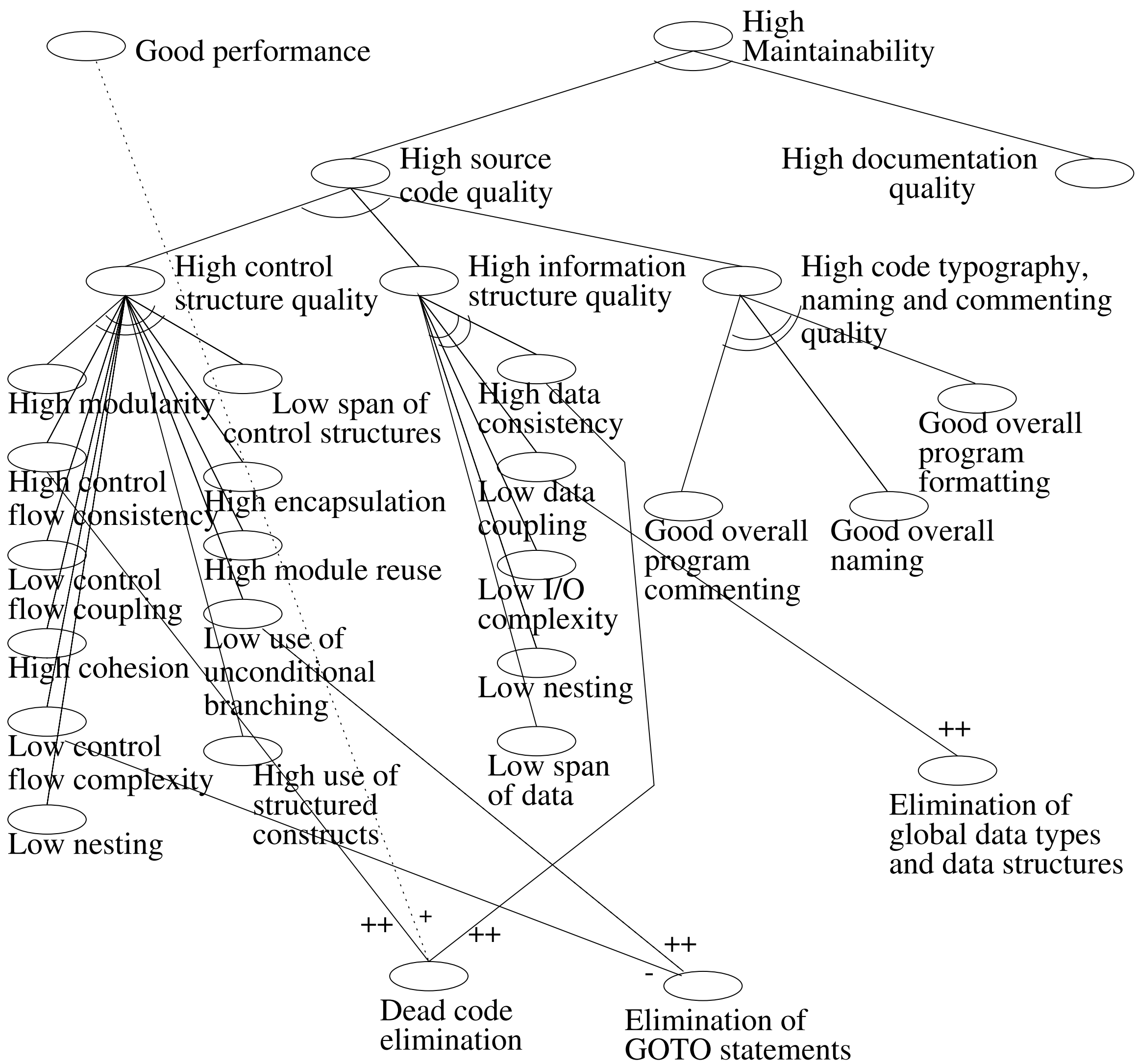

Figure 2.3: Maintainability softgoal interdependency graph, including heuristics

2.3.2 Identifying Heuristics to Satisfice Performance

In this section we briefly describe some of the *heuristics* that can be implemented in a system's source code to satisfice the performance quality requirement. Appendix C provides a full description of all the performance heuristics as well as their contributions, and should be consulted for further details.

The softgoal interdependency graph given in Figure 2.4 illustrates a subset of these heuristics as well as their contributions towards their parent softgoals. A more complete version of this graph illustrating the entire set of heuristics can be found in Figure C.1.

As shown in Figure 2.4, an example of a performance heuristic is *dead code elimination*. This means to eliminate code that is unreachable or that does not affect the program (e.g. dead stores). Implementing this heuristic makes a "+" contribution towards meeting the *low main memory utilization* softgoal, because *dead code elimination* will cause the size of the program to decrease. Implementing this heuristic makes a "+" contribution towards meeting the *low secondary storage utilization* softgoal, because *dead code elimination* will cause the size of the program to decrease. Dead code elimination may also affect maintainability in various ways. We discussed these contributions in the previous section.

As shown in Figure 2.4, another example of a performance heuristic is *elimination of GOTO statements*. This means to minimize the number of GOTO statements in the source code. Implementing this heuristic makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals. Elimination of GOTO statements may also affect maintainability in various ways. We discussed these contributions in the previous section.

As shown in Figure 2.4, another example of a performance heuristic is *integer divide optimization*. This means to replace integer divide instructions with power-of-two denominators and other bit patterns with faster instructions, such as shift instructions. Implementing this heuristic makes a "+" contribution towards meeting the *low user CPU time* softgoal.

A full discussion of the rest of the performance heuristics and their contributions towards their parent softgoals can be found in Appendix C.

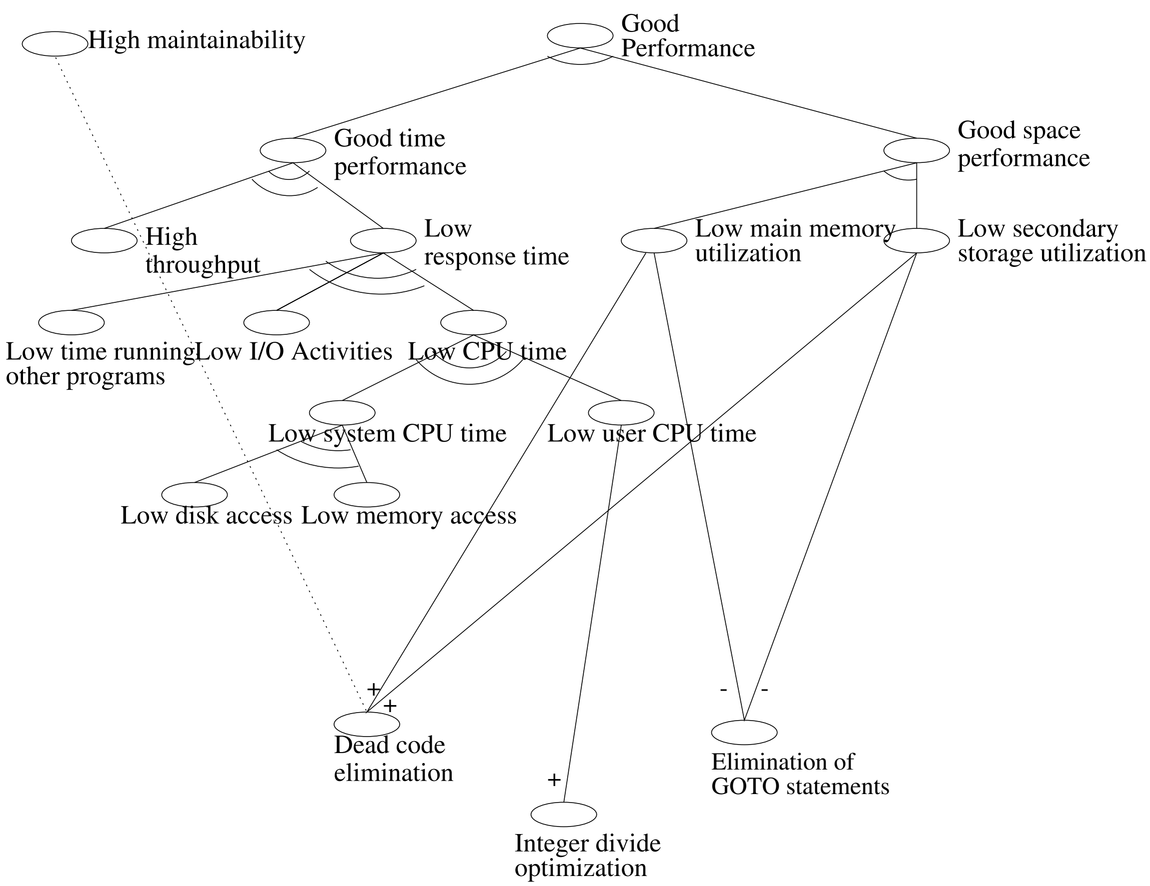

Figure 2.4: Performance softgoal interdependency graph, including heuristics

Chapter 3

Maintainability and Performance Measurements

In this Chapter we perform maintainability and performance optimization activities, by implementing different heuristics at the source code level. Each optimization activity we have performed corresponds directly to a specific heuristic that is described in Appendices B and C.

In each case we evaluated the effect of applying an optimization heuristic on the overall maintainability and performance of the source code, or the overall "code health". In order to estimate the effect of a specific optimization heuristic on the health of source code, a set of metrics were extracted from the code before and after the heuristic was applied. [1]

The C++ source code of two different software systems was modified for our experiments; WELTAB, an election tabulation system, and the AVL GNU tree and linked list libraries. Both systems were originally written in C, but a reengineering tool was used to migrate the procedural C code to the object-oriented C++ language. The primary reason for reengineering WELTAB and AVL from C to C++ was our desire to produce object-oriented code that was of very low quality. This low quality was desirable for our experiments, because it gave us many opportunities to improve the source code by implementing optimization heuristics. Below we provide more details about WELTAB and AVL.

The WELTAB Election Tabulation System was created in the late 1970s to support

[1] Credit is given to Ladan Tahvildari from the University of Waterloo, for her efforts in extracting these source code metrics.

the collection, reporting, and certification of election results by city and county clerks' offices in US. It was originally written in an extended version of Fortran on IBM and Amdahl mainframes under the University of Michigan's MTS operating system. At various times through the 1980s, it was run on Comshare's Commander II time- sharing service on a Xerox Sigma machine, and on IBM 4331 and IPL (IBM 4341 clone) machines under VM/CMS. Each move caused inevitable modifications in the evolution of the code. Later, the system was converted to C and run on PCs under MSDOS (non-GUI, pre-Windows). The latest version of the system is composed of 4.25 KLOC and 35 batch files. Specifically, there are 26 header files, 39 source code files, and the rest are data files for a total of 190 files. For more details on WELTAB, see:

`http://www.darpa.mil/ito/psum1998/D882-0.html`

The GNU AVL Libraries is a public domain library written in C for sparse arrays, AVL, splay trees, and binary search trees. The library also includes code for implementing single and double linked lists. The original system was organized around C structs and a quite elaborate collection of macros for implementing tree traversals, and simulating polymorphic behavior for inserting, deleting and tree re-balancing operations. The system is composed of 4KLOC of C code, distributed in 6 source files and 3 library files. For more details on AVL, see:

`http://ftp.cs.stanford.edu/gnu/avl/`

It is important to note that in this chapter we only discuss a subset of these metrics. A full discussion of all extracted metrics can be found in Appendix D.

3.1 Maintainability Measurements

In order for maintenance processes to be improved and for the amount of effort expended in software maintenance activities to be reduced, it is first necessary to be able to measure software maintainability. [32] In this Section we demonstrate how software maintainability metrics can be used to evaluate the effects of optimizations in the source code. A number of different maintenance and performance optimization activities were applied to the WELTAB and AVL object-oriented C++ software systems.

For each optimization activity, a set of maintainability metrics models were applied to the object-oriented C++ source code, both before and after the optimization activity took place. This analysis of the differences in maintainability measures, before and after some maintainability or performance optimization activity took place, serves two purposes:

1. To evaluate the effect of the maintenance or performance optimization activity on the maintainability of the source code, and

2. To determine how sensitive a particular maintainability metrics model is, to the type of maintenance or performance optimization activity that was performed.

3.1.1 Maintainability Metrics Models

In this section, the most important maintainability metrics that were extracted from the WELTAB and AVL C++ source code are described. It is important to note that for readability purposes, we only describe a subset of the maintainability metrics extracted. A full description of all maintainability metrics can be found in Appendix D. The *MI1*, *MI2* and *MI3* metrics were extracted at both the *file* level and *function* level for each optimization heuristic.

In each case the metrics were extracted automatically using DATRIX, a tool for assessing the software quality of C and C++ systems. DATRIX can automatically extract approximately 110 different metrics on a system's source code, to evaluate how well the system satisfies various software characteristics. For more details on DATRIX, see:

`http://www.iro.umontreal.ca/labs/gelo/datrix/prodinfo/prodinfo.htm`

Maintainability Indexes

MI1

This is a single maintainability index, based on Halstead's metrics. It is computed using the following formula:

$$MI1 = 125 - 10 * LOG(avg - E)$$

The term $avg - E$ is defined as follows:

- avg-E = average Halstead Volume V per module

MI2

This is a single maintainability index, based on Halstead's metrics, McCabe's Cyclomatic Complexity, lines of code and number of comments. It is computed using the following formula:

$$MI2 = 171 - 5.44 * ln(avg - E) - 0.23 * avg - V(G) - 16.2 * ln(avg - LOC)$$

$$+50 * sin(sqrt(2.46 * (avg - CMT/avg - LOC)$$

The coefficients are derived from actual usage.The terms are defined as follows:

- avg-E = average Halstead Volume V per module
- avg-V(G) = average extended cyclomatic complexity per module
- avg-LOC = the average count of lines of code (LOC) per module
- avg-CMT = average percent of lines of comments per module

MI3

This is a single maintainability index, based on Halstead's metrics, McCabe's Cyclomatic Complexity, lines of code and number of comments. It is computed using the following formula:

$$MI3 = 171 - 3.42 * ln(avg - E) - 0.23 * avg - V(G) - 16.2 * ln(avg - LOC)$$

$$+0.99 * avg - CMT$$

The coefficients are derived from actual usage.The terms are defined as follows:

- avg-E = average Halstead Volume V per module
- avg-V(G) = average extended cyclomatic complexity per module
- avg-LOC = the average count of lines of code (LOC) per module
- avg-CMT = average percent of lines of comments per module

3.1.2 A study of the optimization activities

In this section we describe how we conducted pre-post analyses of the maintainability metrics for each of the optimization heuristics.

The pre-post analysis of the maintainability metrics was performed on nine different code optimization heuristics; four of these heuristics focused on improving performance and the other five focused on improving maintainability. Following is a brief description of the performance and maintainability optimization heuristics:

Hoisting and Unswitching - The FOR loops were optimized, so that each iteration executed faster (performance optimization).

Address Optimization - References to global variables that used a constant address were replaced with references using a pointer and offset (performance optimization).

Integer Divide Optimization - Integer divide instructions with power-of-two denominators were replaced with shift instructions, which are faster (performance optimization).

Function Inlining - When a function was called in the program, the body of the function was expanded inline (performance optimization).

Elimination of GOTO statements - The number of GOTO statements in the source code was minimized (maintainability optimization).

Dead Code Elimination - Code that was unreachable or that did not affect the program was eliminated (maintainability optimization).

Elimination of Global Data Types and Data Structures - Global data types and data structures were made local (maintainability optimization).

Maximization of Cohesion - Classes with low cohesion were split into many smaller classes, when possible (maintainability optimization).

Minimization of Coupling Through ADTs - Variables declared within a class, which have a type of ADT which is another class definition, were eliminated (maintainability optimization).

Some of these activities were applied to WELTAB only, others to AVL only, and others to both systems. We first extracted *file* level and *function* level maintainability metrics on the original WELTAB and AVL C++ source code before any of the optimization activities took place. For each distinct performance and maintainability optimization activity, we then extracted *file* level and *function* level maintainability metrics on either WELTAB or AVL or both, after the activity took place.

It is important to note that for both WELTAB and AVL there exist many other optimization activities that could have been applied to the source code. However, the C++ source code of both systems was of such low quality, that it did not allow us to apply many other optimizations that we would have liked to. It was difficult to understand and modify both WELTAB and AVL, since even slight changes could affect other parts of the system in undesirable ways.

The reason for this low quality is that the C++ code was the result of a reengineering effort to migrate the original C version to an object-oriented language. The reengineering tool used for this purpose focused on producing code that was correct rather than readable. Thus, although the resulting C++ versions of WELTAB and AVL executed properly, it was difficult to understand and maintain the new systems.

We now provide a detailed analysis of these performance and maintainability optimization activities, by explaining the pre-post changes in the maintainability metrics.

Hoisting and Unswitching

The objective of this performance optimization activity was to optimize run-time performance by minimizing the time spent during FOR loops.

Hoisting refers to cases where loop-invariant expressions are executed within FOR loops. In such cases, the loop-invariant expressions can be moved out of the FOR loops, thus improving run-time performance by executing the expression only once rather than at each iteration. [16]

For example, in the code fragment below, the expression (x+y) is loop invariant, and the addition can be hoisted out of the loop.

```
for (i = 0; i < 100; i++) {
  a[i] = x + y;
}
```

Below is the code fragment after the invariant expression has been hoisted out of the loop.

```
t = x + y;
for (i = 0; i < 100; i++) {
  a[i] = t;
}
```

Unswitching refers to transforming a FOR loop containing a loop-invariant IF statement into an IF statement containing two FOR loops. [16]

For example, in the code fragment below, the IF expression is loop-invariant, and can be hoisted out of the loop.

```
for (i = 0; i < 100; i++)
  if (x)
    a[i] = 0;
  else
    b[i] = 0;
```

After unswitching, the IF expression is only executed once, thus improving run-time performance.

```
if (x)
  for (i = 0; i < 100; i++)
    a[i] = 0;
else
  for (i = 0; i < 100; i++)
    b[i] = 0;
```

This heuristic was implemented in WELTAB only. Measurements were taken at both the *file* level and the *function* level. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table 3.1.

All the Maintainability Indexes (MIs) decreased. These descreases can be attributed to the fact that all Halstead's metrics and lines of code (variables that affect the MIs) increased (see Appendix D for details). Thus, *Hoisting and Unswitching* had as a result that maintainability was affected negatively in the optimized system.

Metric	Pre-Value	Post-Value
MI1	71.9263	71.9256
MI2	36.6910	36.6757
MI3	61.3768	61.3618

Table 3.1: File level maintainability metrics on the WELTAB system before and after hoisting/unswitching

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.2. All those measurements also show a decrease in maintainability after hoisting/unswitching.

Function	Metric	Pre-Value	Post-Value
report-canv	MI1	63.18	63.18
	MI2	-16.50	-16.50
	MI3	12.26	12.26
Baselib-smove	MI1	86.55	85.36
	MI2	75.09	70.87
	MI3	92.97	89.31

Table 3.2: Function level maintainability metrics on the WELTAB system before and after hoisting/unswitching

Integer Divide Optimization

The objective of this performance optimization activity was to replace integer divide expressions with power-of-two denominators with faster integer shift instructions. [16]

For example, the integer divide expression in the code fragment below can be replaced with a shift expression:

```
int f (unsigned int i)
{
  return i / 2;
}
```

Below is the code fragment after the integer divide expression has been replaced with a shift expression:

```
int f (unsigned int i)
{
  return i >> 1;
}
```

This heuristic was implemented in both WELTAB and AVL. In WELTAB measurements were taken at both the *file* level and the *function* level. In AVL measurements were taken at the *function* level only. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table 3.3.

Metric	Pre-Value	Post-Value
MI1	71.9263	71.9256
MI2	36.6910	36.6902
MI3	61.3768	61.3763

Table 3.3: File level maintainability metrics on the WELTAB system before and after integer divide optimization

It is interesting to observe that most of the metrics did not change at all, and even those that did changed only slightly. These measures alone show that the new optimized system is almost as maintainable as the original one. However, we know that the new system is less maintainable because some divide instructions of the original system got replaced with shift instructions which are less intuitive.

All the Maintainability Indexes (MIs) decreased slightly. Thus, *Integer Divide Optimization* had as a result that maintainability was affected negatively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.4, and on the optimized version of AVL in Table 3.5. All those measurements also show a decrease in maintainability after integer divide optimization.

Address Optimization

The objective of this performance optimization activity was to fit all the global scalar variables of WELTAB in a global variable pool. Then, each of the global scalar variables

Function	Metric	Pre-Value	Post-Value
wcre-showdone	MI1	70.05	69.90
	MI2	22.44	22.25
	MI3	48.00	47.88
weltab-showdone	MI1	70.05	69.91
	MI2	22.44	22.27
	MI3	48.00	47.89

Table 3.4: Function level maintainability metrics on the WELTAB system before and after integer divide optimization

Function	Metric	Pre-Value	Post-Value
ubi_cacheGet	MI1	88.40	88.04
	MI2	87.16	86.71
	MI3	104.19	103.90

Table 3.5: Function level maintainability metrics on the AVL system before and after integer divide optimization

gets accessed via one pointer and an offset, instead of via constant address. This way, more expensive load and store sequences are avoided and code size is reduced. [16]

This is an example of how the global variables were declared and referenced in the original WELTAB system:

```
int nwrite;
int untspilt;
int untavcbs;
int untstart;
int untnprec;
int untwards;
int unitno;

void f (void)
{
  unitno = 10;
```

```
    return;
  }
```

Below is the new code fragment after the global variables got mapped into a global memory pool. As we can see, the global variable *unitno* is now referenced by adding an offset 6 to the pointer *AddressOpt*.

```
  int AddrOpt[7];
  int *AddressOpt = &AddrOpt[0];

  void f (void)
  {
    *(AddressOpt+6) = 10;
    return;
  }
```

This heuristic was implemented in WELTAB only. Measurements were taken at both the *file* level and the *function* level. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table 3.6.

Metric	Pre-Value	Post-Value
MI1	71.9263	71.8982
MI2	36.6910	36.6559
MI3	61.3768	61.3547

Table 3.6: File level maintainability metrics on the WELTAB system before and after address optimization

All the Maintainability Indexes (MIs) decreased. These descreases can be attributed to the fact that all Halstead's metrics (variables that affect the MIs) increased (see Appendix D). Thus, *Address Optimization* had as a result that maintainability was affected negatively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.7. All those measurements also show a decrease in maintainability after address optimization.

Function	Metric	Pre-Value	Post-Value
cmprec-xfix	MI1	62.39	62.37
	MI2	-18.10	-18.13
	MI3	11.03	11.01
cmprec-prec	MI1	67.49	67.46
	MI2	11.60	11.55
	MI3	38.35	38.32
cmprec-vedt	MI1	62.29	62.26
	MI2	-18.78	-18.81
	MI3	10.39	10.37
cmprec-vset	MI1	75.88	75.89
	MI2	41.99	42.00
	MI3	64.84	64.84
cmprec-vfix	MI1	62.45	62.42
	MI2	-17.06	-17.09
	MI3	12.04	12.02
files-rsprtpag	MI1	65.23	65.22
	MI2	1.74	1.73
	MI3	29.54	29.54
files-prtpag	MI1	65.20	65.19
	MI2	1.62	1.60
	MI3	29.43	29.42
report-fixw	MI1	75.56	75.57
	MI2	40.88	40.89
	MI3	63.87	63.88
report-cmut	MI1	70.77	70.78
	MI2	21.93	21.93

continued on next page

continued from previous page

Function	Metric	Pre-Value	Post-Value
	MI3	47.15	47.15
report-chead	MI1	81.41	81.41
	MI2	62.78	62.78
	MI3	83.05	83.05
report-rsum	MI1	68.48	68.48
	MI2	13.74	13.75
	MI3	40.03	40.03
report-lans	MI1	67.99	67.99
	MI2	11.23	11.23
	MI3	37.75	37.75
report-cnv1a	MI1	64.20	64.13
	MI2	-10.32	-10.41
	MI3	17.96	17.91
report-canv	MI1	63.18	63.12
	MI2	-16.50	-16.58
	MI3	12.26	12.21
weltab-sped	MI1	68.32	68.25
	MI2	9.82	9.74
	MI3	36.19	36.14
weltab-poll	MI1	64.70	64.66
	MI2	-4.10	-4.15
	MI3	23.95	23.92
weltab-spol	MI1	63.64	63.60
	MI2	-10.60	-10.64
	MI3	17.94	17.91
weltab-getprec	MI1	79.08	78.63
	MI2	56.93	56.36

continued on next page

continued from previous page			
Function	Metric	Pre-Value	Post-Value
	MI3	78.29	77.93
weltab-pget	MI1	64.15	63.73
	MI2	-6.30	-6.82
	MI3	22.00	21.67
weltab-showpoll	MI1	67.49	67.36
	MI2	15.32	15.16
	MI3	42.07	41.97
weltab-showdone	MI1	70.05	69.91
	MI2	22.44	22.27
	MI3	48.00	47.89
weltab-allowcard	MI1	73.18	73.12
	MI2	34.66	34.59
	MI3	58.77	58.72

Table 3.7: Function level maintainability metrics on the WELTAB system before and after address optimization

Function Inlining

The objective of this performance optimization activity was to eliminate the overhead associated with calling and returning from a function, by expanding the body of the function inline.

For example, in the code fragment below, the function add() can be expanded inline at the call site in the function sub().

```
int add (int x, int y)
{
  return x + y;
}

int sub (int x, int y)
{
  return add (x, -y);
}
```

Expanding add() at the call site in sub() yields:

```
int sub (int x, int y)
{
  return x + -y;
}
```

Function inlining usually increases code space, which is affected by the size of the inlined function, and the number of call sites that are inlined.

This heuristic was implemented in both WELTAB and AVL. In WELTAB measurements were taken at both the *file* level and the *function* level. In AVL measurements were taken at the *function* level only. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table 3.8.

Metric	Pre-Value	Post-Value
MI1	71.9263	71.4982
MI2	36.6910	35.5612
MI3	61.3768	60.4460

Table 3.8: File level maintainability metrics on the WELTAB system before and after function inlining

All the Maintainability Indexes (MIs) decreased. These descreases can be attributed to the fact that all Halstead's metrics and lines of code (variables that affect the MIs) increased (see Appendix D). Thus, *Function Inlining* had as a result that maintainability was affected negatively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.9, and on the optimized version of AVL in Table 3.10. All those measurements also show a decrease in maintainability after function inlining.

Function	Metric	Pre-Value	Post-Value
weltab-poll	MI1	64.70	64.19

continued on next page

continued from previous page

Function	Metric	Pre-Value	Post-Value
	MI2	-4.10	-4.33
	MI3	23.95	20.95
weltab-spol	MI1	63.64	63.21
	MI2	-10.60	-11.56
	MI3	17.94	15.18
report-cand	MI1	80.68	80.68
	MI2	56.09	56.09
	MI3	76.71	76.71
report.rsum	MI1	68.48	67.94
	MI2	13.74	12.00
	MI3	40.03	38.54
report-cnv1a	MI1	64.20	61.66
	MI2	-10.32	-11.30
	MI3	17.96	16.16
report-canvw	MI1	77.14	75.11
	MI2	46.06	39.07
	MI3	68.32	62.27
report-dhead	MI1	78.83	73.16
	MI2	52.48	44.72
	MI3	73.96	68.83
report-canv	MI1	63.18	61.48
	MI2	-16.50	-17.20
	MI3	12.26	9.34
Baselib-setdate	MI1	88.86	71.99
	MI2	85.25	64.20
	MI3	102.06	72.86
Baselib-cvec	MI1	79.81	76.68

continued on next page

continued from previous page			
Function	Metric	Pre-Value	Post-Value
	MI2	56.15	48.85
	MI3	77.16	66.33

Table 3.9: Function level maintainability metrics on the WELTAB system before and after function inlining

Function	Metric	Pre-Value	Post-Value
ubi_btInsert	MI1	77.85	77.73
	MI2	47.39	47.24
	MI3	69.32	69.22
ubi_cache Delete	MI1	91.18	90.59
	MI2	94.48	93.76
	MI3	110.22	109.76
ubi_cache Reduce	MI1	91.96	91.32
	MI2	93.33	92.53
	MI3	108.70	108.19
ubi_cacheSet MaxEntries	MI1	92.79	87.15
	MI2	101.13	88.93
	MI3	116.14	106.58
ubi_cacheSet MaxMemory	MI1	92.79	87.15
	MI2	101.16	88.98
	MI3	116.14	106.58
ubi_cachePut	MI1	91.44	84.88
	MI2	91.20	79.57
	MI3	106.81	98.23

Table 3.10: Function level maintainability metrics on the AVL system before and after function inlining

Elimination of GOTO statements

The objective of this maintenance optimization activity was to minimize the number of GOTO statements in WELTAB. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.

It is important to note that the original WELTAB C++ source code contained a very large number of GOTO statements. It was not possible to eliminate all GOTO statements, since in many cases removing them would have altered the source code's control flow. Each GOTO statement that was eliminated got replaced with a block of executable statements, ending with a return statement. Thus, it was ensured that the control flow in the optimized version was exactly the same as in the original version of WELTAB.

This heuristic was implemented in WELTAB only. Measurements were taken at both the *file* level and the *function* level. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table 3.11.

Metric	Pre-Value	Post-Value
MI1	71.9263	71.6085
MI2	36.6910	35.4542
MI3	61.3768	60.2877

Table 3.11: File level maintainability metrics on the WELTAB system before and after eliminating GOTO statements

It is important to note that maintainability did get improved by eliminating GOTO statements. Elimination of GOTO statements is the only way to minimize the number of unconditional branches in source code. Decreasing the number of unconditional branches is a key factor in improving maintainability, as it can assist a maintainer in understanding the source code of a system. [6] In our measurements, the number of unconditional branches is shown by the metric RtnGotoNbr, which decreased significantly after GOTO statements were eliminated.

However, elimination of GOTO statements also affects other characteristics of source code in varying ways, and thus maintainability may get affected in different ways. After eliminating GOTO statements many of the DATRIX measurements showed that source code became slightly less maintainable. These measurements are shown in Table 3.11.

All the Maintainability Indexes (MIs) decreased. These descreases can be attributed to the fact that all Halstead's metrics, McCabe's Cyclomatic Complexity and lines of code (variables that affect the MIs) increased (see Appendix D).

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.12. All those measurements show a decrease in maintainability.

Function	Metric	Pre-Value	Post-Value
weltab-sped	MI1	68.32	67.44
	MI2	9.82	5.22
	MI3	36.19	31.99
weltab-poll	MI1	63.64	63.87
	MI2	-10.60	-6.72
	MI3	17.94	21.72
weltab-spol	MI1	63.64	62.85
	MI2	-10.60	-13.07
	MI3	17.94	15.83
weltab-allowcard	MI1	73.18	72.83
	MI2	34.66	33.70
	MI3	58.77	57.96
cmprec-xfix	MI1	62.45	62.04
	MI2	-17.06	-19.00
	MI3	12.04	10.28
cmprec-vfix	MI1	62.45	62.09
	MI2	-17.06	-18.01
	MI3	12.04	11.24
cmprec-vset	MI1	75.88	75.11
	MI2	41.99	39.24
	MI3	64.84	62.45
cmprec-vedt	MI1	62.29	61.94
	MI2	-18.78	-19.72
	MI3	10.39	9.61
cmprec-prec	MI1	67.49	67.36
	MI2	11.60	10.81
	MI3	38.35	37.62
report-cnv1a	MI1	64.20	63.96

continued on next page

continued from previous page			
Function	Metric	Pre-Value	Post-Value
	MI2	-10.32	-10.72
	MI3	17.96	17.67
report-cmut	MI1	70.77	70.62
	MI2	21.93	21.46
	MI3	47.15	46.75
report-fixw	MI1	75.56	74.94
	MI2	40.88	39.25
	MI3	63.87	62.53

Table 3.12: Function level maintainability metrics on the WELTAB system before and after eliminating GOTO statements

Dead Code Elimination

The objective of this maintenance optimization activity was to eliminate dead code that was unreachable or that did not affect the program. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.

It is important to note that the original WELTAB C++ source code contained a large amount of dead code. It cannot be certain that all dead code was eliminated. However, after dead code was eliminated on some source files, the size of the files decreased by almost half their original size. This fact alone points out the importance of dead code elimination, not only for maintainability purposes, but also for space performance purposes.

This heuristic was implemented in WELTAB only. Measurements were taken at both the *file* level and the *function* level. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table 3.13.

All the Maintainability Indexes (MIs) increased significantly, by nearly 30%. These increases can be attributed to the fact that all Halstead's metrics (variables that affect the MIs) decreased (see Appendix D). Thus, *Dead Code Elimination* had as a result that maintainability was affected positively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.14. All those measurements also show an increase in maintainability

Metric	Pre-Value	Post-Value
MI1	71.9263	77.2713
MI2	36.6910	56.6653
MI3	61.3768	78.8650

Table 3.13: File level maintainability metrics on the WELTAB system before and after eliminating dead code

after eliminating dead code.

Elimination of Global Data Types and Data Structures

The objective of this maintenance optimization activity was to turn global data types and data structures to local. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.

This heuristic was implemented in WELTAB only. Measurements were taken at both the *file* level and the *function* level. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table 3.15.

All the Maintainability Indexes (MIs) increased. These increases can be attributed to the fact that all Halstead's metrics (variables that affect the MIs) decreased (see Appendix D). Thus, *Elimination of Global Data Types and Data Structures* had as a result that maintainability was affected positively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.16. All those measurements also show an increase in maintainability after eliminating global data types and data structures.

Maximization of Cohesion

The objective of this maintenance optimization activity was to split a class with low cohesion into many smaller classes, each of which has higher cohesion. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.

This heuristic was implemented in AVL only, and measurements were taken at the *function* level only. The *function* level measurements taken on the new optimized version

Function	Metric	Pre-Value	Post-Value
report	MI1	70.43	76.32
	MI2	36.22	55.32
	MI3	61.43	73.67
card	MI1	72.76	73.23
	MI2	38.32	49.23
	MI3	62.78	71.06
weltab	MI1	70.23	75.98
	MI2	39.03	49.32
	MI3	61.43	77.32
files	MI1	69.45	74.32
	MI2	40.01	56.98
	MI3	62.67	78.02
cmprec	MI1	68.04	72.76
	MI2	36.43	51.56
	MI3	64.98	77.32

Table 3.14: Function level maintainability metrics on the WELTAB system before and after eliminating dead code

of AVL are shown in Table 3.17. All those measurements show an increase in maintainability after maximizing cohesion.

Minimization of Coupling Through ADTs

The objective of this maintenance optimization activity was to eliminate variables declared within a class, which have a type of ADT that is another class definition. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.

This heuristic was implemented in AVL only, and measurements were taken at the *function* level only. The *function* level measurements taken on the new optimized version of AVL are shown in Table 3.18. All those measurements show an increase in maintainability after minimizing coupling through ADTs.

Metric	Pre-Value	Post-Value
MI1	71.9263	71.9391
MI2	36.6910	36.7616
MI3	61.3768	61.4414

Table 3.15: File level maintainability metrics on the WELTAB system before and after eliminating global data types and data structures

Function	Metric	Pre-Value	Post-Value
report	MI1	71.92	81.02
	MI2	36.69	38.91
	MI3	61.38	62.04
weltab	MI1	73.18	74.56
	MI2	38.55	39.76
	MI3	65.44	65.59

Table 3.16: Function level maintainability metrics on the WELTAB system before and after eliminating global data types and data structures

3.1.3 Some conclusions on measuring maintainability

In this study, we have studied the maintainability of a software system by extracting a variety of metrics using the DATRIX tool. We did not follow the traditional approach to measuring maintainability, which is to use a single metrics model (such as the MI). One of the disadvantages associated with this traditional approach is that it gives a single index of maintainability. This single index may not represent maintainability as accurately as all the individual metrics taken together do. Thus, examining only a single index could be a mistake. By looking only at a single value you miss the detailed information provided by the variety of metrics we have taken, which permit you to understand the nature of the maintenance activities that took place. [32]

It appears from the results of our experiments that a single index would not have been sensitive to the types of changes that took place. For example, in the case of *Elimination of GOTO statements* most of the metrics did not measure any improvements, although it is well known that this heuristic improves the maintainability of software systems.

Another case where metrics failed to represent maintainability accurately was in the

Function	Metric	Pre-Value	Post-Value
SampleRec	MI1	93.65	94.66
	MI2	103.03	105.01
	MI3	119.21	121.89

Table 3.17: Function level maintainability metrics on the AVL system before and after maximizing cohesion

case of the *Integer Divide Optimization* heuristic. One could argue that metrics did not change significantly because the maintainability of the source code did not change. However, maintainability got affected negatively, since we replaced divide instructions with shift instructions.

Some studies in this section showed the failings of using a single measure of maintainability. Obviously there is more to source code maintainability than just lines of code and number of comments. These results suggest that a good maintainability assessment tool should not only provide a simplistic index of maintainability, but it should also provide other raw metrics that are necessary to interpret and understand that index. A single maintainability index may serve only as a rough estimate of the maintainability of the source code under study. [32] In order for someone to keep track of a good combination of all software attributes that affect maintainability, it is necessary to examine a separate metric for each attribute. [6]

3.2 Performance Measurements

In order for the performance of a software system to be improved, it is first necessary to be able to measure software performance. [32] In this section we demonstrate how software performance measurements were used to evaluate the effects of specific changes to a system's source code. A number of different maintenance and performance optimization heuristics were applied to one or both of the WELTAB and AVL C++ software systems. For each activity, performance measurements were taken at the *function-level* both before and after the planned activity took place.

Function	Metric	Pre-Value	Post-Value
ubi_cacheRoot	MI1	76.86	79.31
	MI2	98.77	102.67
	MI3	108.44	111.45
ubi_idbDB	MI1	83.46	85.18
	MI2	88.67	93.63
	MI3	99.46	106.32
ubi_btNode	MI1	92.76	96.17
	MI2	92.49	93.25
	MI3	116.21	117.38
ubi_idb FuncRec	MI1	81.07	88.93
	MI2	107.33	117.43
	MI3	127.32	139.87

Table 3.18: Function level maintainability metrics on the AVL system before and after minimizing coupling through ADTs

3.2.1 A study of the optimization activities

In this section we describe the pre-post analysis of the performance measurements for each of the optimization activities.

The pre-post analysis of performance measurements was performed on most of the optimization heuristics that were presented in Section 3.1. For each distinct optimization heuristic, we extracted performance measurements on WELTAB and/or AVL both before and after the heuristic was applied. Performance measurements were taken only at the *function-level*.

There exist many other performance optimization activities that could have been implemented in WELTAB as well. However, the C++ source code was of such low quality that it did not allow us to implement many of the other performance activities that we would have liked to. It was difficult to understand and modify WELTAB, since even slight changes could affect other parts of the system in undesirable ways.

We next describe for each optimization activity the pre-post changes in the performance measurements that took place.

Hoisting and Unswitching

The objective of this performance optimization activity was to optimize run-time performance by minimizing the time spent during FOR loops. For more details on the actual heuristic, see Section 3.1.2.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.19. As we can see, this heuristic was applied to 2 different locations of the source code. In both cases, performance was improved because of the heuristic. Thus, we can say with confidence that this heuristic affected time performance positively.

Function in WELTAB system	Performance of the original function	Performance after hoisting and unswitching
report-canv	0.32	0.28
Baselib-smove	0.83	0.69

Table 3.19: Function level performance metrics on the WELTAB system before and after hoisting and unswitching

Integer Divide Optimization

The objective of this performance optimization activity was to replace integer divide expressions with power-of-two denominators with faster integer shift instructions. For more details on the actual heuristic, see Section 3.1.2.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.20, and on the new optimized version of AVL in Table 3.21. As we can see, in all cases performance was improved because of the heuristic. Thus, we can say with confidence that this heuristic affected time performance positively.

Address Optimization

The objective of this performance optimization activity was to fit all the global scalar variables of WELTAB in a global variable pool. Then, each of the global scalar variables gets accessed via one pointer and an offset, instead of via constant address. This way,

Function in WELTAB system	Performance of the original function	Performance after integer divide optimization
wcre-showdone	0.76	0.65
weltab-showdone	0.33	0.28

Table 3.20: Function level performance metrics on the WELTAB system before and after integer divide optimization

Function in AVL system	Performance of the original function	Performance after integer divide optimization
ubi_cacheGet	0.45	0.43

Table 3.21: Function level performance metrics on the AVL system before and after integer divide optimization

more expensive load and store sequences are avoided and code size is reduced. [16] For more details on the actual heuristic, see Section 3.1.2.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.22. As we can see, this heuristic was applied to many different locations of the source code. Performance was improved in all cases. Thus, we can say with confidence that this heuristic affected time performance positively.

Function Inlining

The objective of this performance optimization activity was to eliminate the overhead associated with calling and returning from a function, by expanding the body of the function inline. For more details on the actual heuristic, see Section 3.1.2.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.23, and on the new optimized version of AVL in Table 3.24. As we can see, this heuristic was applied to many different locations of the source code. Performance was improved in all cases. Thus, we can say with confidence that this heuristic affected time performance positively.

Elimination of GOTO statements

The objective of this maintenance activity was to minimize the number of GOTO statements in WELTAB. For more details on the actual heuristic, see Section 3.1.2.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.25. As we can see, this heuristic was applied to multiple different locations of the source code. Performance was improved in some case, and was affected negatively in other cases. Thus, the results do not provide sufficient evidence that elimination of GOTO statements affects performance in a specific way. Performance may be affected differently, depending on the method used to eliminate GOTO statements.

Dead Code Elimination

The objective of this maintenance optimization activity was to eliminate dead code that was unreachable or that did not affect the program. For more details on the actual heuristic, see Section 3.1.2.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table 3.26. As we can see, this heuristic was applied to 5 different locations of the source code. In almost all cases, performance was improved because of the heuristic. Thus, we can say with confidence that this heuristic affected performance positively.

Elimination of Global Data Types and Data Structures

The objective of this maintenance optimization activity was to turn global data types and data structures to local. For more details on the actual heuristic, see Section 3.1.2.

The *function level* measurements taken on the new optimized version of WELTAB are shown in Table 3.27. As we can see, this heuristic was applied to 2 different locations of the source code. In both cases, performance was hurt. Thus, we can say with confidence that this heuristic affected performance negatively.

Maximization of Cohesion

The objective of this maintenance optimization activity was to split a class with low cohesion into many smaller classes, each of which has higher cohesion. For more details on the actual heuristic, see Section 3.1.2.

The *function* level measurements taken on the new optimized version of AVL are shown in Table 3.28. As we can see, this heuristic was applied to 1 source code location and performance was affected negatively.

Minimization of Coupling Through ADTs

The objective of this maintenance optimization activity was to eliminate variables declared within a class, which have a type of ADT that is another class definition. For more details on the actual heuristic, see Section 3.1.2.

The *function* level measurements taken on the new optimized version of AVL are shown in Table 3.29. As we can see, this heuristic was applied to 4 source code locations and performance was hurt in all cases. Thus, we can say with confidence that this heuristic affected performance negatively.

Function in WELTAB system	Performance on the original function	Performance after address optimization
cmprec-xfix	0.32	0.31
cmprec-prec	0.76	0.71
cmprec-vedt	0.11	0.07
cmprec-vset	0.19	0.18
cmprec-vfix	0.98	0.87
files-rsprtpag	0.32	0.26
files-prtpag	0.41	0.35
report-fixw	0.32	0.29
report-cmut	0.41	0.39
report-chead	0.76	0.63
report-rsum	0.44	0.45
report-lans	0.87	0.86
report-cnv1a	0.54	0.53
report-canv	0.32	0.27
weltab-sped	0.65	0.61
weltab-poll	0.32	0.31
weltab-spol	0.98	0.97
weltab-getprec	0.87	0.85
weltab-pget	0.43	0.41

Table 3.22: Function level performance metrics on the WELTAB system before and after address optimization

Function in WELTAB system	Performance on the original function	Performance after function inlining
weltab-poll	0.81	0.42
weltab-spol	0.32	0.23
report-cand	0.87	0.78
report-rsum	0.43	0.32
report-cnv1a	0.99	0.88
report-canvw	0.28	0.23
report-dhead	0.76	0.65
report-canv	0.87	0.73
Baselib-setdate	0.54	0.41
Baselib-cvec	0.87	0.72

Table 3.23: Function level performance metrics on the WELTAB system before and after function inlining

Function in AVL system	Performance on the original function	Performance after function inlining
ubi_btInsert	0.03	0.02
ubi_cache-Delete	0.13	0.10
ubi_cache-Reduce	0.21	0.19
ubi_cacheSet-MaxEntries	0.32	0.31
ubi_cacheSet-MaxMemory	0.77	0.73
ubi_cachePut	0.58	0.55

Table 3.24: Function level performance metrics on the AVL system before and after function inlining

Function in WELTAB system	Performance on the original function	Performance after elimination of GOTO statements
weltab-sped	0.12	0.23
weltab-poll	0.13	0.17
weltab-spol	0.03	0.04
weltab-allowcard	0.32	0.33
cmprec-xfix	0.23	0.24
cmprec-vfix	0.31	0.35
cmprec-vset	0.12	0.32
cmprec-vedt	0.51	0.50
cmprec-prec	0.76	0.81
report-cnv1a	0.43	0.42
report-cmut	0.21	0.35
report-fixw	0.41	0.39

Table 3.25: Function level performance metrics on the WELTAB system before and after elimination of GOTO statements

Function in WELTAB system	Performance on the original function	Performance after dead code elimination
report	0.45	0.44
card	0.33	0.31
weltab	0.69	0.61
files	0.32	0.28
cmprec	0.76	0.77

Table 3.26: Function level performance metrics on the WELTAB system before and after dead code elimination

Function in WELTAB system	Performance on the original function	Performance after elimination of global data types and data structures
report	0.21	0.22
weltab	0.78	0.79

Table 3.27: Function level performance metrics on the WELTAB system before and after elimination of global data types and data structures

Function in AVL system	Performance on the original function	Performance after maximizing cohesion
SampleRec	0.67	0.69

Table 3.28: Function level performance metrics on the AVL system before and after maximizing cohesion

Function in AVL system	Performance on the original function	Performance after minimizing coupling
ubi_cacheRoot	0.67	0.68
ubi_idbDB	0.56	0.58
ubi_btNode	0.45	0.49
ubi_idbFuncRec	0.73	0.74

Table 3.29: Function level performance metrics on the AVL system before and after minimizing coupling

Chapter 4

Selecting a Heuristic Transformation

During the course of our experiments, we realised that the effectiveness of an optimization heuristic in improving a system's quality depends upon some of the system's specific characteristics. When using the NFR framework to select a set of optimization heuristics, such characteristics are not being taken into account. However, a developer should take these software characteristics into account, when choosing the set of optimization heuristics to be implemented in a system.

Specifically, for any candidate optimization heuristic a software developer should examine:

- the number of source code locations to which the heuristic can be applied, and
- the chances that these source code locations will be maintained during the maintenance process (for a maintainability optimization heuristic) or executed during run-time (for a performance optimization heuristic).

For example, a performance optimization heuristic may be very effective if:

- it can be applied to many source code locations, or
- it can be applied to source code locations that get executed frequently during run-time.

The $80-20$ rule is often used to describe such situations [15]. This rule states that 20% of the source code will be executed 80% of the time; and similarly that 20% of the source code will be maintained 80% of the time. Thus, in selecting the best combination of optimization heuristics, a developer should attempt to select heuristics that can be applied to many source code locations falling under the $80-20$ category.

The following formula, which we will refer to as *Andre formula*, should be used in conjunction with a *softgoal interdependency graph* for a particular software quality:

$$(x_1 + log x_2) * improvement$$

where

x_1 is the number of source code locations that fall under the $80 - 20$ category, to which the heuristic under consideration can be applied.

x_2 is the number of source code locations that do not fall under the $80 - 20$ category, to which the heuristic under consideration can be applied.

improvement is an integer representing the developer's *subjective* estimation of the heuristic's quality, leaving aside any system characteristics that may affect the heuristic's effectiveness.

The purpose of the *Andre formula* is to assist a developer in selecting the optimization heuristics that will improve software quality the most. It allows for a developer to take into consideration the software characteristics that will affect the optimization's effectiveness in a particular situation. Such software characteristics include the number of source code locations to which the optimization heuristic can be applied, that fall under the $80 - 20$ category. Of course there is some subjectivity involved in using this formula; but such subjectivity is unavoidable because it is impossible to draw a clear-cut line between the source code locations falling under the $80 - 20$ category and those not.

4.1 Validation of the Andre Formula

We tested the *Andre formula* to show that it gives a reliable indication of the best set of optimization heuristics. The formula was tested on the maintainability optimization heuristics that were implemented in WELTAB during our experiments.

The results of our tests are shown in Table 4.1. The first step was to apply *Andre formula* on the dead code elimination heuristic, because all our measurements showed that this heuristic had the best overall effect on the maintainability of WELTAB. The next step was to apply Andre formula on other maintainability optimization heuristics that resulted in a smaller benefit for WELTAB.

As shown in Table 4.1, the formula resulted in a higher value for the heuristics that truly had the best overall effect on the maintainability of WELTAB.

Optimization heuristic	x_1	x_2	Improvement	Result
Dead Code Elimination	2	3	3	7.4313638
Elimination of Global Data Types and Data Structures	1	1	1	1
Maximization of Cohesion	0	1	2	0
Minimization of Coupling	2	2	2	4.60206

Table 4.1: Testings that show the reliability of Andre formula

Chapter 5

Conclusions

The main goal of this report was to propose a framework for driving the software reengineering process on the basis of quality requirements. This framework defines and guides the migration of legacy procedural code to an object-oriented language, while maintaining certain qualities to a desirable level. Our framework can also be viewed as a generic methodology for selecting the set of optimization heuristics that will improve the system's software quality the most, while minimizing negative side-effects.

The major contributions of this report include using the *NFR framework* to model two particular software qualities, maintainability and performance. We identified and described many heuristic transformations that affect these software qualities and that can be implemented in a target system's source code.

We also presented an evaluation procedure for experimentally evaluating the effect of heuristic transformations on software quality. This evaluation procedure can be used to determine the set of optimization heuristics that will maximize the benefit on the system, while minimizing negative side-effects.

Finally, we conducted experiments by implementing some of the heuristic transformations in two medium-sized software systems and then collecting measurements. The experimental results justify our proposed contributions of heuristic transformations towards software quality.

5.1 Future Work

The most important problem faced is the lack of standardized software metrics, to assess the degree to which a quality requirement is satisficed by a set of heuristics. As DeMarco pointed out, "you cannot control what you cannot measure." The quality of software products cannot be controlled, unless that quality can first be measured; and software metrics are the only means known to measure software quality. [28, 29]

Unfortunately, software metrics have not been studied adequately, especially in the object-oriented paradigm. Few metrics have been proposed to measure object-oriented systems, and even those have not been validated properly. Thus, software quality can not be measured precisely and more research is still required in the field of software metrics.

A research direction for further investigation, is the possibility to use our NFR models for maintainability and performance as software metrics models. This could provide a big advantage over the metrics models that already exist, because our NFR models allow one to consider software characteristics that previous metrics models ignored. For example, our NFR models permit one to assign different weights to the various software characteristics that affect maintainability. The weighted contributions of all software characteristics could then be summed in a formula, to create a measurement indicating the degree to which maintainability has been achieved in a system.

Furthermore, our NFR models for maintainability and performance could be used for the purpose of validating the existing software metrics models. Our NFR models provide an understanding of what ranges of measurements can be considered reasonable for a specific system. For example, maintainability metrics could be extracted from different versions of a software system, both before and after maintainability optimization heuristics have been applied to the system; if the results of the measurements are consistent with what our NFR models tell us to expect, then we can consider those software metrics to be reliable.

Bibliography

[1] L. Chung, *"Non-Functional Requirements in Software Engineering"*, PhD thesis, University of Toronto, Toronto, Ontario, Canada, 1993.

[2] IEEE, *Standard Glossary of Software Engineering Terminology*, 1983.

[3] IEEE, *Standard Glossary of Software Engineering Terminology*, 1990.

[4] B. Meyer, *Object-Oriented Software Construction* (Prentice Hall, 1988).

[5] I. Sommerville, *Software Engineering* (Addison-Wesley, 1989).

[6] J. R. Hagemeister, "A Metric Approach to Assessing the Maintainability of Software", Master's thesis, University of Idaho, Moscow, Idaho, 1992.

[7] P. Hsia, "A Study on the Effect of Architecture on Maintainability of Object-Oriented Systems", in *Proceedings 1995 International Conference on Software Maintena nce*, pp. 295–303, IEEE CS Press, 1995.

[8] J. Daly, "The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study", in *Proceedings 1995 International Conference on Software Maintena nce*, pp. 154–160, IEEE CS Press, 1995.

[9] P. M. R. Penteado and M. Cagnin, "An Experiment of Legacy Code Segmentation to Improve Maintainability", in *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, pp. 111–119, IEEE CS Press, 1999.

[10] S. M. Lionel C. Briand and V. Basili, "Measuring and Assessing Maintainability at the End of High-Level Design", in *Proceedings IEEE Conference on Software Maintenance 1993 and Tools Fair*, IEEE CS Press, 1993.

[11] P. O. F. Zhuo, B. Lowther and J. Hagemeister, "Constructing and Testing Software Maintainability AssessmentModels", in *Proceedings of the First International Software Metrics Symposium*, pp. 61–70, IEEE CS Press, 1993.

[12] J. A. P. Oman, D. Ash and B. Lowther, "Using Software Maintainability Models to Track Code Health", in *Proceedings of International Conference on Software Maintenance*, pp. 154–160, IEEE CS Press, 1994.

[13] H. M. Sneed and A. Kaposi, "A study on the effect of reengineering on maintainability", in *In Proceedings of the Conference On Software Maintenance*, pp. 91–99, IEEE Computer Society Press, 1990.

[14] S. H. G. A. Kiran and P. Jalote, "Effect of Object Orientation on Maintainability of Software", in *International Conference on Software Maintenance*, pp. 91–99, IEEE Computer Society Press, 1997.

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann, San Mateo, CA, 1990).

[16] D. F. Bacon, *"Fast and Effective Optimization of Statically Typed Object-Oriented Languages"*, PhD thesis, University of California, Berkeley, Berkeley, California, 1997.

[17] B. A. Nixon, Implementation of information system design specifications: A performance perspective, in *Database Programming Languages: Bulk Types and Persistent Data. 3rd International Workshop, August 27-30, 1991, Nafplion, Greece, Proceedings*, edited by P. C. Kanellakis and J. W. Schmidt, pp. 149–168, Morgan Kaufmann, 1991.

[18] B. A. Nixon, Representing and using performance requirements during the development of information systems, in *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, edited by M. Jarke, J. A. B. Jr., and K. G. Jeffery, , Lecture Notes in Computer Science Vol. 779, pp. 187–200, Springer, 1994.

[19] J. Arthur and K. Stevens, "Assessing the Adequacy of Documentation Through Document Quality Indicators", in *Proceedings Conference on Software Maintenance*, pp. 40–49, IEEE CS Press, 1989.

[20] R. M. Baecker and A. Marcus, *Human Factors and Typography for More Readable Programs* (Addison Wesley, 1989).

[21] P. W. Oman and C. R. Cook, Typographic style is more than cosmetic Vol. 33, pp. 506–520, Communications of the ACM (CACM), 1990.

[22] R.Penteado, P.Masiero, and M.Cagnin, An experiment of legacy code segmentation to improve maintainability, Third European Conference on Software Maintenance and Reengineering, 1999.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison Wesley, 1995).

[24] T. Korson and V. Vaishnavi, "An Empirical Study of the Effects of Modularity on Program Modifiability", in *Empirical Studies of Programmers*, pp. 168–186, Ablex Publishing Corp., 1986.

[25] Federal Information Processing Standards(FIPS), *Guideline on Software Maintenance*, 1997.

[26] L. Briand, C. Bunse, J. Daly, and C. Differing, "An experimental comparison of the maintainability of object-oriented and structured design documents", in *Proceedings International Conference on Software Maintenance*, IEEE CS Press, 1997.

[27] M. Shepperd and D. C. Ince, A critique of three metrics Vol. 26, pp. 197–210, 1994.

[28] H. D. Rombach, Impact of software structure on maintenance, pp. 152–160, IEEE, 1985.

[29] H. D. Rombach, A controlled experiment on the impact of software structure on maintainability, pp. 344–354, TSE, 1987.

[30] L. XuanDong and Z. GuoLiang, Enhancing reusability and maintainability in ndoom, pp. 236–246, OOIS, 1997.

[31] P. Oman and J. Hagemeister, Construction and testing of polynomials predicting software maintainability, pp. 251–??, 1994.

[32] T. Pearse and P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities", in *Proceedings 1995 International Conference on Software Maintenance*, pp. 295–303, IEEE CS Press, 1995.

[33] H.Sneed and A.Merey, Automated software quality assurance, IEEE Transactions on Software Engineering, 1985.

Appendix A

Glossary

Cohesion : "module strength; the manner and degree to which the tasks performed by a single software module are related to one another." [6]

Control flow complexity : "the degree to which a system has a design or implementation that is difficult to understand and verify." [6]

Control flow consistency : "the degree of uniformity, standardization, and freedom from contradiction of the logical process flow within the parts of a system or component." [6]

Control flow coupling : "the manner and degree of interdependence between software modules. Types include common-environment, content, control, data, hybrid, pathological." [6]

Control Structure : "characteristics affecting the choice and use of control flow constructs, the manner in which the system or program is decomposed into algorithms, and the method in which those algorithms are implemented." [6]

CPU time : the component of response time which the CPU spends working on our behalf; the time since a program started, during which the program was using the CPU; the total direct CPU cost of executing the program. CPU time is composed of user CPU time and system CPU time. CPU time excludes time spent waiting for I/O or time running other programs; it also excludes the CPU costs of parts of the kernel that run on behalf of the program. For example, the cost of stealing page frames to replace the page frames taken from the free list when the program started is not reported as part of the program's CPU time. [15]

Data consistency : "the degree of uniformity, standardization, and freedom from contradiction among the intermodular data types and structures of a system." [6]

Data coupling : "the manner and degree of interdependence between software modules. Types include common-environment, content, control, data, hybrid, pathological." [6]

Encapsulation : "a software development technique that consists of isolating a system function or a set of data, and operations on those data, within a module and providing precise specifications for the module." [6]

Information Structure : "characteristics affecting the choice and use of data structure and data flow techniques. The manner in which information is stored and manipulated throughout the system or program." [6]

I/O activity : Abbreviation of input/output activity; an activity of transferring data to and from peripheral devices such as hard disks, tape drives, the keyboard, and the screen. During the execution of a program, I/O activities may be required to bring in the program's text and data, or to acquire real memory for the program's use.

I/O complexity : "the degree of complication of a system component, determined by factors such as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures, and other local characteristics." [6]

Maintainability: "The characteristics of the software, its history, and associated environments that affect the maintenance process and are indicative of the amount of effort necessary to perform maintenance changes. It can be measured as a quantification of the time necessary to make maintenance changes to the product." [3, 6]

Maintenance: "The process of implementing corrective, adaptive, or perfective software changes." [6]

Modularity : "the degree to which a system or program is composed of discrete components such that a change to one component has minimal impact on other components." [6]

Module reuse : "the degree to which a software module can be used in more than one location in a program or system." [6]

Nesting : "to place subroutines/data in other subroutines/data at a different hierarchical level so that subroutines/data can be executed/accessed recursively; to incorporate program constructs into other constructs." [6]

Overall naming : "the name, address, label, or distinguishing index of objects in a computer program." [6]

Overall program commenting: "information embedded within a computer program that provides clarification to human readers but does not affect machine interpretation." [6]

Overall program formatting: "the use of typography and commenting to make a program appear more elegant and easier to read."[6]

Performance: can be defined in terms of speed (time performance) or it can be defined in terms of storage capacity (space performance). [15]

Response time: the total time to complete a task; the elapsed time from beginning to end of a program. Response time is composed of CPU time, I/O activity time, and time consumed by other programs. [15]

Space performance: a general term referring to the storage requirements of a program. [15]

Span of control structures : "the number of statements contained within a given control statement in which operations are performed." [6]

Span of data : "the number of statements between the first and last references of that variable." [6]

Structured construct : "a control structure having one entry and one exit. May be a sequence of two or more instructions, a conditional selection of one of two or more sequences of instructions, or a repetition of a sequence of instructions." [6]

System CPU time : the CPU time spent in the operating system performing tasks on behalf of the program; the time used by system calls invoked by a program (directly or indirectly). [15]

Throughput: "the total amount of work done by a computer in a given time." [15]

Time performance: a general term referring to the speed of a program; can be defined in terms of throughput or response time. [15]

Typography, Naming, and Commenting : "characteristics affecting the typographic layout, naming and commenting of code. These characteristics have no effect on program execution, but they affect program comprehension and, therefore, maintenance." [6]

Unconditional branching : "a jump that takes place regardless of execution conditions." [6]

User CPU time : the CPU time spent in the program; time used by a program itself and any library subroutines it calls. [15]

Appendix B

Description of Maintainability Optimization Heuristics

Tables B.1-B.33 give all the *heuristics* that we are aware of, that can be implemented in a system's source code to contribute towards satisficing the maintainability quality requirement. These tables explain the heuristics (if necessary), and also discuss the contributions that each heuristic makes towards satisficing its parent softgoals. Each table also gives the rationale underlying the heuristic's contributions towards parent softgoals.

The softgoal interdependency graph given in Figure B.1 illustrates all these heuristics and their contributions towards their parent softgoals.

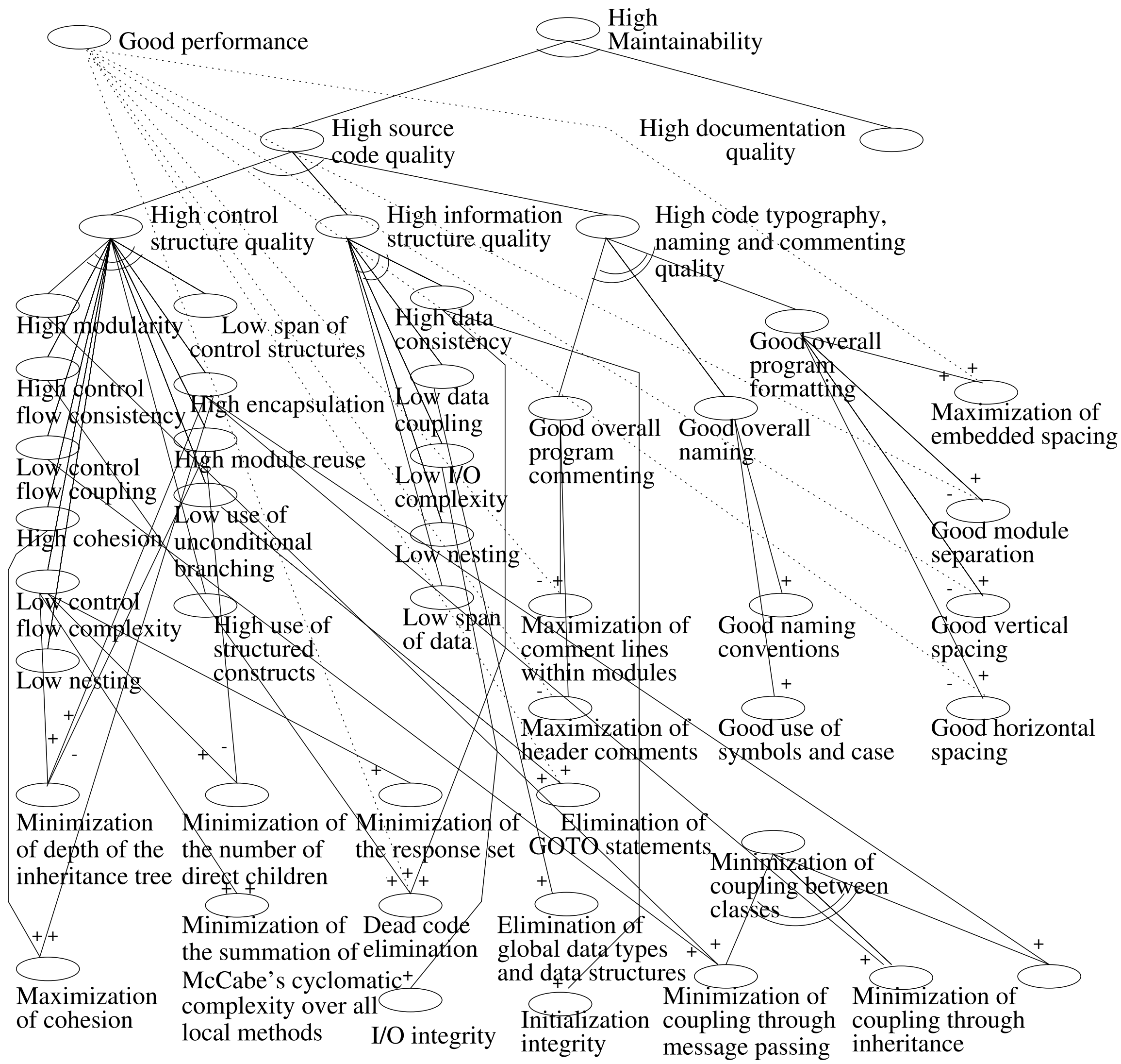

Figure B.1: Maintainability softgoal interdependency graph, including heuristics

Heuristic	Explanations (if required)	Contributions and Rationale
Minimization of the depth of the inheritance tree	Minimize the position of a class in the inheritance hierarchy.	Implementing this heuristic makes a "+" contribution towards meeting the *low control flow complexity* softgoal, because the less decendants a class has, the less classes it may potentially affect because of inheritance (for example, by modifying methods or instance variables defined in the superclass). Implementing this heuristic also makes a "+" contribution towards meeting the *high encapsulation* softgoal, because the lower a class is in the inheritance tree, the more superclass properties this class may access because of its inheritance. If the subclass accesses the inherited properties from the superclass without using the methods defined in the superclass, then encapsulation of the superclass is violated. However, implementing this heuristic makes a "-" contribution towards meeting the *high module reuse* softgoal, because the higher a class is in the inheritance tree, the less superclass properties this class may access because of its inheritance. Thus, it may need to redefine properties defined in other classes.

Table B.1: Minimization of the depth of the inheritance tree

Heuristic	Explanations (if required)	Contributions and Rationale
Minimization of the number of direct children for a class		Implementing this heuristic makes a "+" contribution towards meeting the *low control flow complexity* softgoal, because the more direct children a class has, the more classes it may potentially affect because of inheritance. For example, if there are many subclasses of the class that are dependent on some methods or instance variables defined in the superclass, any changes to these methods or variables may affect the subclasses. Then complexity will be affected negatively. However, implementing this heuristic makes a "-" contribution towards meeting the *high module reuse* softgoal, because the less direct children a class has, the less classes will reuse the properties that have already been defined. Then module reuse will be affected negatively.

Table B.2: Minimization of the number of direct children

Heuristic	Explanations (if required)	Contributions and Rationale
Minimization of the response set for a class.	This means to minimize for each class the number of its local methods, as well as the number of calls to other methods from local methods.	Implementing this heuristic makes a "+" contribution towards meeting the *low control flow complexity* softgoal, because the larger the response set for a class, the larger are the number of methods that get called in response to a message. Then complexity (which is defined as "the degree to which a system has a design or implementation that is difficult to understand and verify, determined by factors such as the number and intricacy of interfaces" [6]) will be affected negatively. One may also intuit that a class with a high response set is hard to maintain, because calling a large number of methods in response to a message makes tracing an error difficult.

Table B.3: Minimization of the response set

Heuristic	Explanations (if required)	Contributions and Rationale
Maximization of cohesion for a class.	The cohesion of a class is characterized by how closely the local methods are related to the local instance variables in the class. A class has low cohesion if it has many disjoint sets of local methods. A disjoint set of local methods is a collection of local methods that do not intersect with each other. Any two local methods do not intersect with each other, if they access at least one common local instance variable. [3]	Implementing this heuristic makes a "+" contribution towards meeting the *high cohesion* and *high encapsulation* softgoals. The rationale behind these contributions is that if all the methods defined in a class access many independent sets of data structures encapsulated in the class, then encapsulation could be increased by splitting the class into many other classes. Thus, a class with low cohesion is not well partitioned and designed and thus is hard to maintain.

Table B.4: Maximization of cohesion

Heuristic	Explanations (if required)	Contributions and Rationale
Minimization of the summation of McCabe's cyclomatic complexity over all local methods for a class.	This can be done either by minimizing the number of local methods of a class, or by minimizing the McCabe's cyclomatic complexity of each individual local method.	Implementing this heuristic makes a "++" contribution towards meeting the *low control flow complexity* softgoal, because the complexity for a system can be measured, among other things, by "McCabe's cyclomatic complexity averaged over all modules." [6] The more methods a class has, the higher McCabe's cyclomatic complexity for that class will be. Similarly, the more control flows a class's methods have, the higher McCabe's cyclomatic complexity for that class will be. Thus, it will be harder to understand the classes and harder to maintain them.

Table B.5: Minimization of the summation of McCabe's cyclomatic complexity over all local methods

Heuristic	Explanations (if required)	Contributions and Rationale
Dead code elimination	This means to eliminate code that is unreachable or that does not affect the program (e.g. dead stores).	Implementing this heuristic makes a "++" contribution towards meeting the *high control flow consistency* softgoal, because control flow consistency is measured by the "percent of code anomalies, where percent of code anomalies is the number of lines of dead code divided by the size of the system." [6] Implementing this heuristic also makes a "++" contribution towards meeting the *high data consistency* softgoal, because data consistency is measured by the "percent of data flow anomalies, where percent of data flow anomalies is the number of data flow anomalies (used before definition, definition without use, redefinition without use) divided by the total number of data structures". [6]

Table B.6: Dead code elimination

Heuristic	Explanations (if required)	Contributions and Rationale
Elimination of GOTO statements	This means to minimize the number of GOTO statements in the source code.	Implementing this heuristic makes a "++" contribution towards meeting the *low use of unconditional branching* softgoal, because both unconditional branches and GOTO statements can be defined as "a jump that takes place regardless of execution conditions". [6] Implementing this heuristic also makes a "-" contribution towards meeting the *low control flow complexity* softgoal, because it was proved in our experiments that eliminating GOTO statements may make the source code more complex.

Table B.7: Elimination of GOTO statements

Heuristic	Explanations (if required)	Contributions and Rationale
Elimination of global data types and data structures		Implementing this heuristic makes a "++" contribution towards meeting the *low data coupling* softgoal. The rationale behind this contribution is that data coupling is measured, among other things, by "the number of global structures and passed parameters divided by the total number of data structures." [6] More specifically, a global data type or data structure can be accessed by two or more modules of a program without being explicitly passed as parameters between the modules. Thus, the degree of interdependence between modules increases, and then data coupling increases as well.

Table B.8: Elimination of global data types and data structures

Heuristic	Explanations (if required)	Contributions and Rationale
Initialization integrity	This means to initialize a variable, register, or other storage location to a starting value prior to use. [3]	Implementing this heuristic makes a "+" contribution towards meeting the *high data consistency* softgoal.

Table B.9: Initialization integrity

Heuristic	Explanations (if required)	Contributions and Rationale
I/O integrity	This means to verify input data items before processing them and to confirm the validity of output data before it is transmitted to the external environment. [33]	Implementing this heuristic makes a "++" contribution towards meeting the *low I/O complexity* softgoal.

Table B.10: I/O integrity

Heuristic	Explanations (if required)	Contributions and Rationale
Minimization of coupling between classes.	Two objects are coupled if they act on each other. Certain types of object coupling are provided by the object-oriented paradigm. The types of object coupling are coupling through message passing, coupling through inheritance, and coupling through abstract data types. In cases like this one, the initial *heuristic* may not be specific enough. In these cases, it needs to be further refined and elaborated. Since in the NFR framework we treat heuristics as softgoals, we are able to decompose these heuristics into more specific heuristics, using the same systematic framework that we used for top-level quality requirements. This heuristic can be detailed by decomposing it into any one of the following heuristics: *minimize coupling through message passing*, *minimize coupling through inheritance*, *minimize coupling through abstract data types* This decomposition is shown in Figure 2.3 . The OR contribution joining these three softgoals means that any of the offspring heuristics can be implemented to achieve the parent softgoal.	

Table B.11: Minimization of coupling between classes

Heuristic	Explanations (if required)	Contributions and Rationale
Minimization of coupling through message passing	This means to eliminate local methods of a class calling methods or instance variables of other classes.	Implementing this heuristic makes a "+" contribution towards meeting the *low control flow coupling* softgoal, because if a local method calls many methods or instance variables of other classes, then the implementation of that local method is very dependent on the methods of other classes. Implementing this heuristic also makes a "+" contribution towards meeting the *high modularity* softgoal, because if a local method calls many methods or instance variables of other classes, then the modularity rule that "every module should communicate with as few others as possible" [4] is violated.

Table B.12: Minimization of coupling through message passing

Heuristic	Explanations (if required)	Contributions and Rationale
Minimization of coupling through inheritance	This means to eliminate local methods of a class accessing nonprivate attributes of its superclasses.	Implementing this heuristic makes a "+" contribution towards meeting the *high encapsulation* softgoal. The rationale behind this contribution is that if properties which are encapsulated in a superclass are exposed to a subclass for less restrictive access, then encapsulation and information hiding are violated. The use of inheritance that is not well designed makes the system more complex.

Table B.13: Minimization of coupling through inheritance

Heuristic	Explanations (if required)	Contributions and Rationale
Minimization of coupling through abstract data types	This means to eliminate variables declared within a class, which have a type of ADT which is another class definition.	Implementing this heuristic makes a "+" contribution towards meeting the *high encapsulation* softgoal. The rationale behind this contribution is that if the programming language permits direct access to the private properties of the ADT, then encapsulation is violated.

Table B.14: Minimization of coupling through abstract data types

Heuristic	Explanations (if required)	Contributions and Rationale
Maximization of embedded spacing within the modules.	This means to increase the percent of blank lines within the modules of the program.	Implementing this heuristic makes a "+" contribution towards meeting the *good overall program formatting* softgoal. The rationale behind this contribution is that overall program formatting is measured, among other things, by the "percent of blank lines in the whole program, percent of modules with blank lines, percent of modules with embedded spacing." [6] However, this heuristic also makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals, because increasing the blank lines may result in a larger program.

Table B.15: Maximization of embedded spacing

Heuristic	Explanations (if required)	Contributions and Rationale
Good module separation	This means to prescribe a disciplined uniform approach to the manner in which modules are visually delineated for a reader. One approach is to put white-space before/after the first/last line of each module. [3]	Implementing this heuristic makes a "+" contribution towards meeting the *good overall program formatting* softgoal. The rationale behind this contribution is that overall program formatting is measured, among other things, by the "percent of blank lines in the whole program, percent of modules with blank lines, percent of modules with embedded spacing." [6] However, this heuristics makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals, because increasing the separation between modules will result in a larger program.

Table B.16: Good module separation

Heuristic	Explanations (if required)	Contributions and Rationale
Good vertical spacing	This means to use blank lines or page breaks to act as separators which distinguish different program statements or parts of the program. [5]	Implementing this heuristic makes a "+" contribution towards meeting the *good overall program formatting* softgoal. The rationale behind this contribution is that overall program formatting is measured, among other things, by the "percent of blank lines in the whole program, percent of modules with blank lines, percent of modules with embedded spacing." [6] However, this heuristic makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals, because increasing the spacing between parts of the program will result in a larger program.

Table B.17: Good vertical spacing

Heuristic	Explanations (if required)	Contributions and Rationale
Good horizontal spacing	This means to use indentation, embedded spacing, tabbing and alignment to act as separators which distinguish different parts of the program or parts of a statement. [5]	Implementing this heuristic makes a "+" contribution towards meeting the *good overall program formatting* softgoal. The rationale behind this contribution is that overall program formatting is measured, among other things, by the "percent of blank lines in the whole program, percent of modules with blank lines, percent of modules with embedded spacing." [6] However, this heuristic makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals, because increasing the spacing between parts of the program will result in a larger program.

Table B.18: Good horizontal spacing

Heuristic	Explanations (if required)	Contributions and Rationale
Maximization of comment lines within the modules	This means to maximize the information embedded within all modules, that provides clarification to human readers but does not affect machine interpretation. The initial heuristic is not specific enough. In these cases, it needs to be further refined and elaborated. Since in the NFR framework we treat heuristics as softgoals, we are able to decompose these heuristics into more specific heuristics, using the same systematic framework that we used for top-level quality requirements. This heuristic can be detailed by decomposing it into any one of the following heuristics: *Comment vague code*, *Comment each variable, type, or constant declaration*, *Appropriate length of comments*. The AND contribution joining these three softgoals means that all of the offspring heuristics must be implemented to achieve the parent softgoal.	

Table B.19: Maximization of comment lines within the modules

Heuristic	Explanations (if required)	Contributions and Rationale
Comment vague code	This means to use descriptive comments to clarify vague code, when the programmer's thinking is not obvious from the code (especially when vague code is necessary for performance reasons, to take advantage of machine or operating system features, to maintain consistency within code being modified, etc.)	Implementing this heuristic makes a "++" contribution towards meeting the *good overall program commenting* softgoal. The rationale behind this contribution is that overall program commenting is measured, among other things, by the "percent of comment lines in the whole program". [6] However, this heuristic makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals, because maximization of comments will result in a larger program.

Table B.20: Comment vague code

Heuristic	Explanations (if required)	Contributions and Rationale
Comment each Variable, Type, or Constant Declaration		Implementing this heuristic makes a "++" contribution towards meeting the *good overall program commenting* softgoal. The rationale behind this contribution is that overall program commenting is measured, among other things, by the "percent of comment lines in the whole program". [6] However, this heuristic makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals, because maximization of comments will result in a larger program.

Table B.21: Comment each Variable, Type, or Constant Declaration

Heuristic	Explanations (if required)	Contributions and Rationale
Appropriate Length of Comments	This means to mak the length of the comments appropriate for the complexity of the code being described.	Implementing this heuristic makes a "+" contribution towards meeting the *good overall program commenting* softgoal. The rationale behind this contribution is that overall program commenting is measured, among other things, by the "percent of comment lines in the whole program". [6]

Table B.22: Appropriate Length of Comments

Heuristic	Explanations (if required)	Contributions and Rationale
Maximization of the modules with header (prologue) comments	This means to maximize the information outside all modules, that describes the individual modules but does not affect machine interpretation. The initial heuristic is not specific enough. In these cases, it needs to be further refined and elaborated. Since in the NFR framework we treat heuristics as softgoals, we are able to decompose these heuristics into more specific heuristics, using the same systematic framework that we used for top-level quality requirements. This heuristic can be detailed by decomposing it into any one of the following heuristics: *Include a header comment for each procedure*, *Include a header comment for each file*, *Include a header comment for each logical block or module*. This decomposition is shown in Figure 2.3 . The AND contribution joining these three softgoals means that all of the offspring heuristics must be implemented to achieve the parent softgoal.	

Table B.23: Maximization of the modules with header (prologue) comments

Heuristic	Explanations (if required)	Contributions and Rationale
Include a header comment for each procedure		Implementing this heuristic makes a "++" contribution towards meeting the *good overall program commenting* softgoal. The rationale behind this contribution is that overall program commenting is measured, among other things, by the "percent of modules with header (prologue) comments." [6] However, this heuristic makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals, because increasing the comments will result in an increase in the total size of the program.

Table B.24: Include a header comment for each procedure

Heuristic	Explanations (if required)	Contributions and Rationale
Include a header comment for each file		Implementing this heuristic makes a "++" contribution towards meeting the *good overall program commenting* softgoal. The rationale behind this contribution is that overall program commenting is measured, among other things, by the "percent of modules with header (prologue) comments." [6] However, this heuristic makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals, because increasing the comments will result in an increase in the total size of the program.

Table B.25: Include a header comment for each file

Heuristic	Explanations (if required)	Contributions and Rationale
Include a header comment for each logical block or module		Implementing this heuristic makes a "++" contribution towards meeting the *good overall program commenting* softgoal. The rationale behind this contribution is that overall program commenting is measured, among other things, by the "percent of modules with header (prologue) comments." [6] However, this heuristic makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals, because increasing the comments will result in an increase in the total size of the program.

Table B.26: Include a header comment for each logical block or module

Heuristic	Explanations (if required)	Contributions and Rationale
Good naming conventions	This means to prescribe a uniform approach to assigning the name, address, label, or distinguishing index of an object in a program. [3] The initial heuristic is not specific enough. In these cases, it needs to be further refined and elaborated. Since in the NFR framework we treat heuristics as softgoals, we are able to decompose these heuristics into more specific heuristics, using the same systematic framework that we used for top-level quality requirements. This heuristic can be detailed by decomposing it into any one of the following heuristics: *Meaningful names*, *Reasonable length of names*. This decomposition is shown in Figure 2.3 . The AND contribution joining these two softgoals means that all of the offspring heuristics must be implemented to achieve the parent softgoal.	

Table B.27: Good naming conventions

Heuristic	Explanations (if required)	Contributions and Rationale
Meaningful names	This means to make names of files, procedures, variables, parameters, constants, types, etc. descriptive and meaningful.	Implementing this heuristic makes a "++" contribution towards meeting the *good overall naming* softgoal, because meaningful naming is necessary for prescribing a uniform approach to naming throughout the program; and prescribing a uniform approach is necessary for naming to be consistent throughout the program.

Table B.28: Meaningful names

Heuristic	Explanations (if required)	Contributions and Rationale
Reasonable length of names	This means to avoid names longer than 20 characters.	Implementing this heuristic makes a "++" contribution towards meeting the *good overall naming* softgoal, because names that are too long are difficult to understand.

Table B.29: Reasonable length of names

Heuristic	Explanations (if required)	Contributions and Rationale
Good use of symbols and case	This means to prescribe a uniform approach to the use of visual beacons in identifiers (e.g. embedding " – " or "_" symbols in identifiers and mixing upper and lower case characters in identifiers). [20, 3] The initial heuristic is not specific enough. In these cases, it needs to be further refined and elaborated. Since in the NFR framework we treat heuristics as softgoals, we are able to decompose these heuristics into more specific heuristics, using the same systematic framework that we used for top-level quality requirements. This heuristic can be detailed by decomposing it into any one of the following heuristics: *Form procedure names with words or abbreviations separated by underscores and use mixed case (e.g., Get_Temp)*, *Form variable names, class names, and object names with words and abbreviations using mixed case but no underscores (e.g., SensorTemp)*, *Form names of constants and type definitions using all upper case and using underscores as word separators.* This decomposition is shown in Figure 2.3 . The AND contribution joining these three softgoals means that all of the offspring heuristics must be implemented to achieve the parent softgoal.	

Table B.30: Good use of symbols and case

Heuristic	Explanations (if required)	Contributions and Rationale
Form procedure names with words or abbreviations separated by underscores and use mixed case (e.g., Get_Temp)		Implementing this heuristic makes a "++" contribution towards meeting the *Good overall naming* softgoal, because visual beacons will ease comprehension of the identifiers.

Table B.31: Form procedure names with words or abbreviations separated by underscores and use mixed case (e.g., Get_Temp)

Heuristic	Explanations (if required)	Contributions and Rationale
Form variable names, class names, and object names with words and abbreviations using mixed case but no underscores (e.g., SensorTemp)		Implementing this heuristic makes a "++" contribution towards meeting the *Good overall naming* softgoal, because visual beacons will ease comprehension of the identifiers.

Table B.32: Form variable names, class names, and object names with words and abbreviations using mixed case but no underscores (e.g., SensorTemp)

Heuristic	Explanations (if required)	Contributions and Rationale
Form names of constants and type definitions using all upper case and using underscores as word separators		Implementing this heuristic makes a "++" contribution towards meeting the *Good overall naming* softgoal, because visual beacons will ease comprehension of the identifiers.

Table B.33: Form names of constants and type definitions using all upper case and using underscores as word separators

Appendix C

Description of Performance Optimization Heuristics

Tables C.1-C.30 give all the *heuristics* that we are aware of, that can be implemented in a system to contribute towards satisficing the performance quality requirement. These tables explain the heuristics (if necessary), and also discuss the contributions that each heuristic makes towards satisficing its parent softgoals. Each table also gives the rationale underlying each heuristic's contributions towards its parent softgoals.

The softgoal interdependency graph given in Figure C.1 illustrates all these heuristics and their contributions towards their parent softgoals.

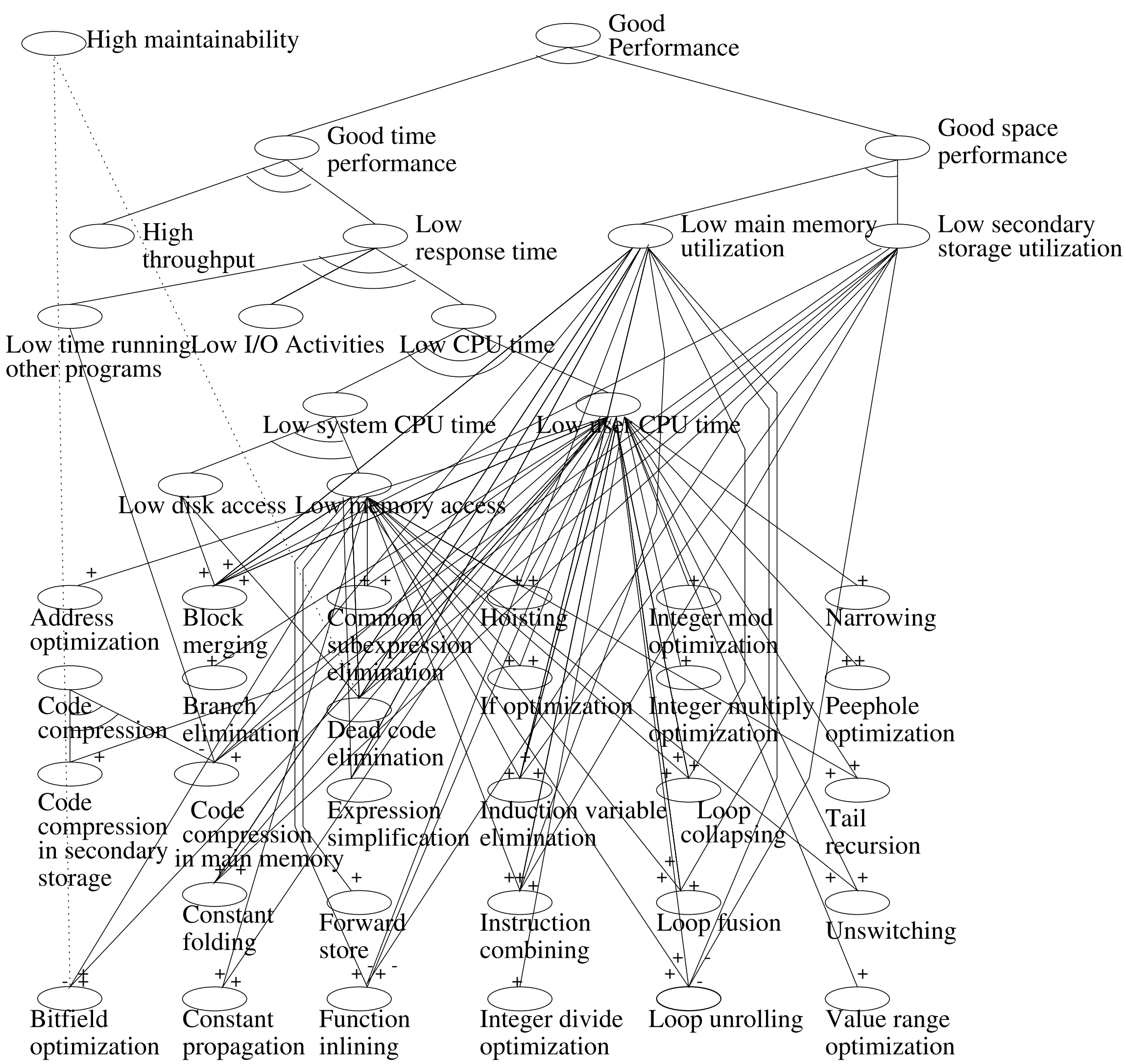

Figure C.1: Performance softgoal interdependency graph, including heuristics

Heuristic	Explanations (if required)	Contributions and Rationale
Address optimization	This means to reference global variables using a pointer and offset, rather than using a constant address. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal. The rationale behind this contribution is that referencing a global variable by constant address requires two instructions, while referencing the same variable through a pointer requires only one. Implementing this heuristic makes a "+" contribution towards meeting the *Low main memory utilization* and *Low secondary storage utilization* softgoals. It was shown in our experiments that Address Optimization may reduce the size of the program, because less space is taken up for variable declarations.

Table C.1: Address optimization

Heuristic	Explanations (if required)	Contributions and Rationale
Bitfield optimization	This means to implement various bitfield optimizations, such as combining adjacent bitfields into one, keeping bitfields in registers, and performing constant propagation through bitfields. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal, because accessing and storing bitfields is expensive, since most architectures do not support bit memory operations and require a series of load/shift/mask/store instructions. Implementing this heuristic also makes a "+" contribution towards meeting the *Low memory access* softgoal, because most architectures do not support bit memory operations and require a series of load/shift/mask/store instructions. However, this heuristic makes a "-" contribution towards meeting the *high data consistency* softgoal, because implementing bitfield optimizations may hurt the degree of uniformity among the data types and structures.

Table C.2: Bitfield optimization

Heuristic	Explanations (if required)	Contributions and Rationale
Block merging	This means to rearrange small blocks of code to create one large basic block. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal. The rationale behind this contribution is that some compilers limit optimizations to basic blocks, and benefit if the program graph can be transformed into a small number of large basic blocks. Implementing this heuristic also makes a "+" contribution towards meeting the *Low main memory utilization* and *Low secondary storage utilization* softgoals. The rationale behind these contributions is that the size of the program may get reduced by replacing many small blocks of code by one large basic block. Implementing this heuristic also makes a "+" contribution towards meeting the *Low disk access* and *Low memory access* softgoals. The rationale behind these contributions is that some compilers limit optimizations to basic blocks, and benefit if the program graph can be transformed into a small number of large basic blocks.

Table C.3: Block merging

Heuristic	Explanations (if required)	Contributions and Rationale
Branch elimination	This means to replace a sequence of two (or more) continuous branches to one branch. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal. The rationale behind this contribution is that with branch elimination less instructions will need to be executed in the program.

Table C.4: Branch elimination

Heuristic	Explanations (if required)	Contributions and Rationale
Code compression	This means to store the executable in compressed form and decompress it during execution. In cases like this one, the initial heuristic is not specific enough, and thus needs to be further refined and elaborated. This heuristic can be detailed by decomposing it into any one of the following heuristics: *Code compression in secondary storage* or *Code compression in main memory*. This decomposition is shown in Figure 2.4. The OR contribution joining these two softgoals means that any of the offspring heuristics can be implemented to achieve the parent softgoal.	

Table C.5: Code compression

Heuristic	Explanations (if required)	Contributions and Rationale
Code compression in secondary storage	This means to store the executable in compressed form in secondary storage and then decompress it as it is being loaded into RAM.	Implementing this heuristic makes a "+" contribution towards meeting the *Low secondary storage utilization* softgoal. The rationale behind this contribution is that secondary storage requirements are reduced, since the executable is stored in compressed form in secondary storage.

Table C.6: Code compression in secondary storage

Heuristic	Explanations (if required)	Contributions and Rationale
Code compression in main memory	This means to store the text portion of the executable in compressed form in RAM, and then decompress it when fetching lines into the instruction cache.	Implementing this heuristic makes a "+" contribution towards meeting the *Low secondary storage utilization* softgoal. The rationale behind this contribution is that secondary storage requirements are reduced, since the executable is stored in compressed form in secondary storage. Implementing this heuristic also makes a "+" contribution towards meeting the *Low main memory utilization* softgoal. The rationale behind this contribution is that program load time and RAM usage are reduced, since the executable is stored in compressed form in RAM. However, this heuristic also makes a "-" contribution towards meeting the *Low user CPU time* and *Low time running other programs* softgoals. The rationale behind these contributions is that this heuristic requires carefully crafted load-time decompression steps, and special software support may be required.

Table C.7: Code compression in main memory

Heuristic	Explanations (if required)	Contributions and Rationale
Constant folding	This means to evaluate expressions with constant operands at compile time. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal, because if run-time evaluation of expressions is avoided then run-time performance will be improved. Implementing this heuristic also makes a "+" contribution towards meeting the *Low main memory utilization* and *Low secondary storage utilization* softgoals. The rationale behind these contributions is that if run-time evaluation of expressions is avoided then code size will be reduced.

Table C.8: Constant folding

Heuristic	Explanations (if required)	Contributions and Rationale
Constant propagation	This means to propagate a constant that is assigned to a variable through the flow graph and substitute it at the use of the variable. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* and *Low memory access* softgoals. The rationale behind these contributions is that by substituting constants with variables at compile-time, less expressions will need to be computed at run-time and less variables will need to get accessed in memory.

Table C.9: Constant propagation

Heuristic	Explanations (if required)	Contributions and Rationale
Common subexpression elimination	This means to avoid recomputing expressions that were previously computed (and whose operands' values have not changed ever since), by using the values of the previous computations. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* and *Low memory access* softgoals. The rationale behind these contributions is that by computing less expressions at run-time less arithmetical operations will occur and less variables will need to get accessed in memory.

Table C.10: Common subexpression elimination

Heuristic	Explanations (if required)	Contributions and Rationale
Dead code elimination	This means to eliminate code that is unreachable or that does not affect the program (e.g. dead stores).	Implementing this heuristic makes a "+" contribution towards meeting the *low main memory utilization* softgoal, because dead code elimination will cause the size of the program to decrease. Implementing this heuristic also makes a "+" contribution towards meeting the *low secondary storage utilization* softgoal, because dead code elimination will cause the size of the program to decrease.

Table C.11: Dead code elimination

Heuristic	Explanations (if required)	Contributions and Rationale
Elimination of GOTO statements	This means to minimize the number of GOTO statements in the source code.	Implementing this heuristic makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals, because it was proved in our experiments that eliminating GOTO statements may cause the size of source code to increase.

Table C.12: Elimination of GOTO statements

Heuristic	Explanations (if required)	Contributions and Rationale
Expression simplification	This means to simplify expressions by replacing them with an equivalent expression that is more efficient. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* and *Low memory access* softgoals. The rationale behind these contributions is that by simplifying expressions less arithmetical operations will occur and thus less variables will need to get accessed in memory.

Table C.13: Expression simplification

Heuristic	Explanations (if required)	Contributions and Rationale
Forward store	This means to move stores to global variables in loops out of the loop, to reduce memory bandwidth requirements. [16]	Implementing this heuristic makes a "+" contribution towards meeting the emphLow memory access softgoal. The rationale behind this contribution is that by moving loads and stores to global variables out of a loop (and keeping values in registers within the loop), less variables will need to get accessed in memory.

Table C.14: Forward store

Heuristic	Explanations (if required)	Contributions and Rationale
Function inlining	This means to expand the body of a function inline, when a function is called in the program. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* and *Low memory access* softgoals. The rationale behind these contributions is that function inlining eliminates the overhead associated with calling and returning from a function. Implementing this heuristic makes a "-" contribution towards meeting the *Low main memory utilization* and *Low secondary storage utilization* softgoals. The rationale behind these contributions is that function inlining usually increases code space, which is affected by the size of the inlined function and the number of inlined functions.

Table C.15: Function inlining

Heuristic	Explanations (if required)	Contributions and Rationale
Hoisting	This means to hoist loop-invariant expressions out of loops. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal, because it will improve run-time performance by executing an expression only once rather than at each iteration. Implementing this heuristic makes a "+" contribution towards meeting the *Low memory access* softgoal, because it will decrease the number of memory accesses by evaluating an expression only once rather than at each iteration.

Table C.16: Hoisting

Heuristic	Explanations (if required)	Contributions and Rationale
If optimization	This means to simplify nested If statements when the value of their conditional expressions are known beforehand. In addition, two adjacent If statements with the same conditional expressions can be combined into one If statement. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* and *Low memory access* softgoals, because less conditional expressions of If statements will need to be evaluated.

Table C.17: If optimization

Heuristic	Explanations (if required)	Contributions and Rationale
Induction variable elimination	This means to combine two or more induction variables within loops, into one induction variable. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal, because by reducing the number of additions or subtractions in a loop run-time performance will improve. Implementing this heuristic makes a "+" contribution towards meeting the *Low memory access* softgoal, because by reducing the number of additions or subtractions in a loop the number of variables that need to get fetched from memory will decrease. Implementing this heuristic makes a "+" contribution towards meeting the *Low main memory utilization* and *Low secondary storage utilization* softgoals, because by reducing the number of additions or subtractions in a loop code space requirements will decrease.

Table C.18: Induction variable elimination

Heuristic	Explanations (if required)	Contributions and Rationale
Instruction combining	This means to combine two statements into one statement, at the source code level. Many operators are candidates for instruction combining, including addition, subtraction, multiplication, left and right shift, boolean operations, and others. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal, because by reducing the number of arithmetical operations run-time performance will improve. Implementing this heuristic makes a "+" contribution towards meeting the *Low memory access* softgoal, because by reducing the number of arithmetical operations the number of variables that need to get fetched from memory will decrease. Implementing this heuristic makes a "+" contribution towards meeting the *Low main memory utilization* and *Low secondary storage utilization* softgoals, because by reducing the number of arithmetical operations code space requirements will decrease.

Table C.19: Instruction combining

Heuristic	Explanations (if required)	Contributions and Rationale
Integer divide optimization	This means to replace integer divide instructions with power-of-two denominators and other bit patterns with faster instructions, such as shift instructions. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal, because on most architectures integer divide instructions are slower than integer shift instructions.

Table C.20: Integer divide optimization

Heuristic	Explanations (if required)	Contributions and Rationale
Integer mod optimization	This means to replace integer modulus instructions with power-of-two operands with faster instructions, such as conditional and shift instructions. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal. The rationale behind this contribution is that the divide and multiply (very slow on most architectures) which are associated with modulus expressions are avoided, and thus run-time performance is increased.

Table C.21: Integer mod optimization

Heuristic	Explanations (if required)	Contributions and Rationale
Integer multiply optimization	This means to replace integer multiply expressions with power-of-two constant multiplicands and other bit patterns with faster instructions, such as shift instructions. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal. The rationale behind this contribution is that on most architectures integer multiply instructions are slower than integer shift instructions.

Table C.22: Integer multiply optimization

Heuristic	Explanations (if required)	Contributions and Rationale
Loop collapsing	This means to collapse nested loops into a single-nested loop. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal, because by reducing loop overhead run-time performance will improve. Implementing this heuristic makes a "+" contribution towards meeting the *Low memory access* softgoal, because by reducing loop overhead the number of variables that get accessed will also be reduced. Implementing this heuristic makes a "+" contribution towards meeting the *Low main memory utilization* softgoal, because by reducing loop overhead the total size of loops will be reduced.

Table C.23: Loop collapsing

Heuristic	Explanations (if required)	Contributions and Rationale
Loop fusion	This means to fuse adjacent loops into one loop. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal, because by reducing loop overhead run-time performance will improve. Implementing this heuristic makes a "+" contribution towards meeting the *Low memory access* softgoal, because by reducing loop overhead the number of variables that get accessed will also be reduced. Implementing this heuristic makes a "+" contribution towards meeting the *Low main memory utilization* softgoal, because by reducing loop overhead the total size of loops will be reduced.

Table C.24: Loop fusion

Heuristic	Explanations (if required)	Contributions and Rationale
Loop unrolling	This means to reduce the number of iterations of a loop by replicating the body of a loop. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal, because by reducing loop overhead run-time performance will improve. Implementing this heuristic makes a "+" contribution towards meeting the *Low memory access* softgoal, because by reducing loop overhead the number of variables that get accessed will also be reduced. However, it makes a "-" contribution towards meeting the *Low main memory utilization* and *Low secondary storage utilization* softgoals, because replicating the bodies of loops will result in a larger program.

Table C.25: Loop unrolling

Heuristic	Explanations (if required)	Contributions and Rationale
Narrowing	This means to use the limited range of small integers to simplify some expressions. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal, because by simplifying expressions less arithmetical operations will occur.

Table C.26: Narrowing

Heuristic	Explanations (if required)	Contributions and Rationale
Peephole optimization	This means to seek to replace short sequences of instructions within a given program with equivalent smaller/faster instruction sequences. This heuristic is typically used only in the final stages of the optimization process, which means that it operates on actual machine instructions as opposed to some higher-level representation of the program. Thus, an implementation of such a heuristic must contain detailed knowledge about the target architecture's instruction set and machine parameters.	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal. The rationale behind this contribution is that by replacing instructions with faster instructions, less CPU time will be required during execution.

Table C.27: Peephole optimization

Heuristic	Explanations (if required)	Contributions and Rationale
Tail recursion	This means to replace a tail-recursive call with a GOTO statement. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* and *Low memory access* softgoals. The rationale behind these contributions is that tail recursion avoids the overhead of a call and return and also reduces stack space usage.

Table C.28: Tail recursion

Heuristic	Explanations (if required)	Contributions and Rationale
Unswitching	This means to transform a loop containing a loop-invariant IF statement into an IF statement containing two loops. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* and *Low memory access* softgoals. The rationale behind these contributions is that since the conditional expression of the IF statement will only be evaluated once, runtime performance will be improved and less variables will need to get fetched from memory.

Table C.29: Unswitching

Heuristic	Explanations (if required)	Contributions and Rationale
Value range optimization	This means to perform optimizations using the known possible range of values of a variable. [16]	Implementing this heuristic makes a "+" contribution towards meeting the *Low user CPU time* softgoal. The rationale behind this contribution is that if the value range optimization involves eliminating expressions, then less arithmetical operations will be performed.

Table C.30: Value range optimization

Appendix D

Maintainability Measurements

This appendix provides a full description of all extracted maintainability metrics. [1]

D.1 Maintainability Metrics Models

In this section, all the maintainability metrics that were extracted from the WELTAB and AVL C++ source code are described in detail. All of these metrics were extracted at the *file* level for each optimization heuristic. In addition, the *MI1*, *MI2* and *MI3* metrics were also extracted at the *function* level for each optimization heuristic.

In each case the metrics were extracted automatically by using the DATRIX tool. The DATRIX tool is a tool used for assessing the software quality of C and C++ systems. DATRIX can automatically extract approximately 110 different metrics on a system's source code, to evaluate how well the system satisfies various software characteristics. The following descriptions of the extracted metrics were taken from the *"DATRIX Metric Reference manual - Version 4"*.

D.1.1 Documentation Metrics

RtnComNbr:

The number of comment sections in the routine's scope (between the routine brackets {...}).

[1] Credit is given to Ladan Tahvildari from the University of Waterloo, for her efforts in extracting these source code metrics.

RtnComVol:

The size in characters of all comments in the routine.

D.1.2 Expression Metrics

RtnCtlCplAvg:

The mean control predicate complexity.

It is computed as the ratio

$$RtnCtlCplSum/(RtnIfNbr + RtnSwitchNbr + RtnLopNbr)$$

$(RtnIfNbr+RtnSwitchNbr+RtnLopNbr)$ represents the number of control transfer statements (decision and loop statements) in the routine.

RtnCtlCplSum:

The sum of the complexities of the control predicates composing the control transfer statements (decision and loop statements) within the routine.

RtnCtlCplMax:

The maximal control predicate complexity.

RtnExeCplAvg:

The mean executable statement complexity.

It is computed as the ratio

$$RtnExeCplSum/RtnExeStmNbr$$

$RtnExeStmNbr$ represents the number of executable statements in the routine.

RtnExeCplSum:

The sum of the complexities of the executable statements within the routine.

RtnExeCplMax:

The maximal executable statement complexity.

D.1.3 General Statement Metrics

RtnStmNbr:

The number of statements in the routine.

RtnXpdStmNbr:

The number of statements after having performed a limited loop unfolding operation where each statement within a loop is taken twice into account (each loop content has been duplicated).

D.1.4 Control-Flow Statement Metrics

RtnCtlStmNbr:

The number of control-flow statements in the routine.

RtnIfNbr:

The number of if statements in the routine.

RtnSwitchNbr:

The number of C-language switch-like constructs in the routine.

RtnLabelNbr:

The number of label statements in the routine.

RtnCaseNbr:

The number of C-language case-like statements in the routine. A C-language case-like statement can only be encountered in a C-language switch-like statement.

RtnDefaultNbr:

The number of default statements in the routine. A default statement can only be encountered in a C-language switch-like statement.

RtnLopNbr:

The number of loop statements in the routine. Loop statements include loop constructs such as for, while, do..while and repeat..until.

RtnReturnNbr:

The number of return statements in the routine.

RtnGotoNbr:

The number of GOTO statements in the routine.

RtnContinueNbr:

The number of continue statements in the routine.

RtnBreakNbr:

The number of break statements in the routine.

D.1.5 Executable Statement Metrics

RtnExeStmNbr:

The number of executable statements in the routine.

RtnSysExitNbr:

The number of system exit call statements in the routine.

D.1.6 Declaration Statement Metrics

RtnDecStmNbr:

The number of declarative statements in the routine.

RtnTypeDecNbr:

The number of type/class declaration statements in the routine.

RtnObjDecNbr:

The number of variable/object declaration statements in the routine.

RtnPrmNbr:

The number of parameters of the routine.

RtnFctDecNbr:

The number of function/routine declaration statements in the routine.

D.1.7 Nesting Level (Scope) Metrics

RtnStmNstLvlSum:

The sum of nesting level values of each statement in the routine. It is used to compute *RtnStmNstLvlAvg*.

RtnStmNstLvlAvg:

The average nesting level of statements in the routine. *RtnStmNstLvlAvg* represents the average nesting level weighted against the number of statements in the routine.

RtnNstLvlMax:

The maximal nesting level in the routine.

RtnScpNstLvlSum:

The sum of nesting level values for all scopes in the routine. A new scope begins whenever an open bracket { is explicitly placed or whenever an implicit (conceptual) open bracket can be deduced, as in:

```
if ( i < 2 ) i++;   (implicit open bracket)
```

RtnScpNstLvlAvg:

The average nesting level of the scopes in the routine. A new scope begins whenever an open bracket { is explicitly placed or whenever an implicit (conceptual) open bracket can be deduced, as in:

```
if ( i < 2 ) i++;   (implicit open bracket)
```

RtnScpNbr:

The total number of scopes in the routine. A new scope begins whenever an open bracket { is explicitly placed or whenever an implicit (conceptual) open bracket can be deduced, as in:

```
if ( i < 2 ) i++;   (implicit open bracket)
```

D.1.8 Cross Reference Metrics

RtnXplCalNbr:

The number of explicit function/method calls in the routine.

RtnXplCastNbr:

The number of explicit type casts in the routine.

D.1.9 McCabe Metric

RtnCycCplNbr:

The cyclomatic number of the routine. The cyclomatic number $v(G)$ was defined by McCabe, and can be computed using the following formula:

$$v(G) = 1 + number\_of\_decision\_points\_in\_the\_routine$$

where a decision point is either:

- an if statement
- a loop statement
- a branch of a switch-like statement (the cases and the default)

D.1.10 Halstead Metrics

OpdNbr:

The total number of operands in the routine's scope.

OpdUnqNbr:

The number of distinct operands in the routine's scope.

OprNbr:

The total number of operators in the routine's scope.

OprUnqNbr:

The number of distinct operators in the routine's scope.

HalDif:

The Halstead program difficulty, for the routine's scope.

HalEff:

The Halstead program effort, for the routine's scope.

HalLen:

The Halstead program length, for the routine's scope.

HalLvl:

The Halstead program level, for the routine's scope.

HalVoc:

The Halstead program vocabulary, for the routine's scope.

HalVol:

The Halstead program volume, for the routine's scope.

D.1.11 Miscellany Metrics

RtnLnsNbr:

The number of lines in the routine.

RtnStxErrNbr:

The number of syntax errors that occurred while parsing the routine.

D.1.12 Maintainability Indexes

MI1:

A single maintainability index, based on Halstead's metrics. It is computed using the following formula:

$$MI1 = 125 - 10 * LOG(avg - E)$$

The term $avg - E$ is defined as follows:

- avg-E = average Halstead Volume V per module

MI2:

A single maintainability index, based on Halstead's metrics, McCabe's Cyclomatic Complexity, lines of code and number of comments. It is computed using the following formula:

$$MI2 = 171 - 5.44 * ln(avg - E) - 0.23 * avg - V(G) - 16.2 * ln(avg - LOC)$$

$$+50 * sin(sqrt(2.46 * (avg - CMT/avg - LOC)$$

The coefficients are derived from actual usage.The terms are defined as follows:

- avg-E = average Halstead Volume V per module
- avg-V(G) = average extended cyclomatic complexity per module
- avg-LOC = the average count of lines of code (LOC) per module
- avg-CMT = average percent of lines of comments per module

MI3:

A single maintainability index, based on Halstead's metrics, McCabe's Cyclomatic Complexity, lines of code and number of comments. It is computed using the following formula:

$$MI3 = 171 - 3.42 * ln(avg - E) - 0.23 * avg - V(G) - 16.2 * ln(avg - LOC)$$

$$+0.99 * avg - CMT$$

The coefficients are derived from actual usage.The terms are defined as follows:

- avg-E = average Halstead Volume V per module
- avg-V(G) = average extended cyclomatic complexity per module
- avg-LOC = the average count of lines of code (LOC) per module
- avg-CMT = average percent of lines of comments per module

D.2 A study of the optimization activities

In this section we describe how we conducted pre-post analyses of the maintainability metrics for each of the optimization activities.

The pre-post analysis of the maintainability metrics was performed on nine different code optimization activities; four of these activities focused on improving performance and the other five focused on improving maintainability. Following is a brief description of the performance and maintainability optimization activities that took place:

Hoisting and Unswitching - The FOR loops were optimized, so that each iteration executed faster (performance optimization).

Address Optimization - References to global variables that used a constant address were replaced with references using a pointer and offset (performance optimization).

Integer Divide Optimization - Integer divide instructions with power-of-two denominators were replaced with shift instructions, which are faster (performance optimization).

Function Inlining - When a function was called in the program, the body of the function was expanded inline (performance optimization).

Elimination of GOTO statements - The number of GOTO statements in the source code was minimized (maintainability optimization).

Dead Code Elimination - Code that was unreachable or that did not affect the program was eliminated (maintainability optimization).

Elimination of Global Data Types and Data Structures - Global data types and data structures were made local (maintainability optimization).

Maximization of Cohesion - Classes with low cohesion were split into many smaller classes, when possible (maintainability optimization).

Minimization of Coupling Through ADTs - Variables declared within a class, which have a type of ADT which is another class definition, were eliminated (maintainability optimization).

Some of these activities were applied to WELTAB only, others to AVL only, and others to both systems. We first extracted *file* level and *function* level maintainability metrics on the original WELTAB and AVL C++ source code before any of the optimization activities took place. For each distinct performance and maintainability optimization activity, we then extracted *file* level and *function* level maintainability metrics on either WELTAB or AVL or both, after the activity took place.

It is important to note that for both WELTAB and AVL there exist many other optimization activities that could have been applied to the source code. However, the C++ source code of both systems was of such low quality, that it did not allow us to apply many other optimizations that we would have liked to. It was difficult to understand and modify both WELTAB and AVL, since even slight changes could affect other parts of the system in undesirable ways.

The reason for this low quality is that the C++ code was the result of a reengineering effort to migrate the original C version to an object-oriented language. The reengineering tool used for this purpose focused on producing code that was correct rather than readable. Thus, although the resulting C++ versions of WELTAB and AVL executed properly, it was difficult to understand and maintain the new systems.

We now provide a detailed analysis of these performance and maintainability optimization activities, by explaining the pre-post changes in the maintainability metrics.

D.2.1 Hoisting and Unswitching

The objective of this performance optimization activity was to optimize run-time performance by minimizing the time spent during FOR loops.

Hoisting refers to cases where loop-invariant expressions are executed within FOR loops. In such cases, the loop-invariant expressions can be moved out of the FOR loops, thus improving run-time performance by executing the expression only once rather than at each iteration. [16]

For example, in the code fragment below, the expression (x+y) is loop invariant, and the addition can be hoisted out of the loop.

```
for (i = 0; i < 100; i++) {
  a[i] = x + y;
}
```

Below is the code fragment after the invariant expression has been hoisted out of the loop.

```
t = x + y;
for (i = 0; i < 100; i++) {
  a[i] = t;
}
```

Unswitching refers to transforming a FOR loop containing a loop-invariant IF statement into an IF statement containing two FOR loops. [16]

For example, in the code fragment below, the IF expression is loop-invariant, and can be hoisted out of the loop.

```
for (i = 0; i < 100; i++)
  if (x)
    a[i] = 0;
  else
    b[i] = 0;
```

After unswitching, the IF expression is only executed once, thus improving run-time performance.

```
if (x)
  for (i = 0; i < 100; i++)
    a[i] = 0;
else
  for (i = 0; i < 100; i++)
    b[i] = 0;
```

This heuristic was implemented in WELTAB only, at both the *file* level and the *function* level. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table D.1.

Metric	Pre-Value	Post-Value
RtnComNbr	0.0000	0.0000
RtnComVol	0.0000	0.0000
RtnCtlCplAvg	4.0483	4.0461
RtnCtlCplSum	64.7692	64.8358
RtnCtlCplMax	9.3003	9.3003
RtnExeCplAvg	7.2738	7.2621
RtnExeCplSum	308.3121	308.4053
RtnExeCplMax	18.2973	18.2973
RtnStmNbr	56.5222	56.5621
RtnXpdStmNbr	99.9482	99.9349
RtnCtlStmNbr	15.3669	15.3802
RtnIfNbr	8.0074	8.0074
RtnSwitchNbr	0.0030	0.0030
RtnLabelNbr	2.4630	2.4630

continued on next page

continued from previous page

Metric	Pre-Value	Post-Value
RtnCaseNbr	0.0266	0.0266
RtnDefaultNbr	0.0000	0.0000
RtnLopNbr	1.6938	1.7071
RtnReturnNbr	1.1036	1.1036
RtnGotoNbr	3.9571	3.9571
RtnContinueNbr	0.5888	0.5888
RtnBreakNbr	0.0133	0.0133
RtnExeStmNbr	32.6672	32.6938
RtnSysExitNbr	0.0000	0.0000
RtnDecStmNbr	8.4882	8.4882
RtnTypeDecNbr	0.0000	0.0000
RtnObjDecNbr	8.4867	8.4867
RtnPrmNbr	2.1553	2.1553
RtnFctDecNbr	0.0015	0.0015
RtnStmNstLvlSum	1.1562	1.1588
RtnStmNstLvlAvg	104.1405	104.2071
RtnNstLvlMax	2.4734	2.4734
RtnScpNstLvlAvg	1.7935	1.7920
RtnScpNstLvlSum	29.7544	29.7811
RtnScpNbr	10.9660	10.9793
RtnXplCalNbr	22.0414	22.0414
RtnXplCastNbr	1.2589	1.2589
RtnCycCplNbr	10.7278	10.7411
OpdNbr	179.5680	179.6346
OpdUnqNbr	47.5769	47.5769
OprNbr	237.6716	237.7382
OprUnqNbr	14.8003	14.8003

continued on next page

continued from previous page		
Metric	Pre-Value	Post-Value
HalDif	23.5483	23.5906
HalEff	202943.3935	202972.7049
HalLen	417.2396	417.3728
HalLvl	0.2173	0.2172
HalVoc	62.3772	62.3772
HalVol	3060.1143	3060.7325
RtnLnsNbr	72.4719	72.5518
RtnStxErrNbr	0.0000	0.0000
MI1	71.9263	71.9256
MI2	36.6910	36.6757
MI3	61.3768	61.3618

Table D.1: File level maintainability metrics on the WELTAB system before and after Hoisting/Unswitching

All of the Halstead base metrics and derived metrics increased. These measures include HalVol, HalLen, HalEff, HalDif, OprNbr, and OpdNbr. These increases can be attributed to the increases in the number of operators and operands. These increases resulted from the fact that in the new optimized version of WELTAB the number of FOR loops has increased.

Other measurements that increased are the number of statements in the routine (RtnStmNbr), the number of control flow statements (RtnCtlStmNbr), and executable statements (RtnExeStmNbr). These increases can also be attributed to the increase in the number of FOR loops in the new version of WELTAB. Thus, it makes sense to conclude that *Unswitching* had a negative effect on main memory utilization.

The final observation we can make is that all the Maintainability Indexes (MIs) decreased. These descreases can be attributed to the fact that all Halstead's metrics and lines of code (variables that affect the MIs) increased. Thus, *Hoisting and Unswitching* had as a result that maintainability was affected negatively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table D.2. All those measurements also show a decrease in maintainability after hoisting/unswitching.

Function	Metric	Pre-Value	Post-Value
report-canv	MI1	63.18	63.18
	MI2	-16.50	-16.50
	MI3	12.26	12.26
Baselib-smove	MI1	86.55	85.36
	MI2	75.09	70.87
	MI3	92.97	89.31

Table D.2: Function level maintainability metrics on the WELTAB system before and after hoisting/unswitching

D.2.2 Integer Divide Optimization

The objective of this performance optimization activity was to replace integer divide expressions with power-of-two denominators with faster integer shift instructions. [16]

For example, the integer divide expression in the code fragment below can be replaced with a shift expression:

```
int f (unsigned int i)
{
  return i / 2;
}
```

Below is the code fragment after the integer divide expression has been replaced with a shift expression:

```
int f (unsigned int i)
{
  return i >> 1;
}
```

This heuristic was implemented in both WELTAB and AVL. In WELTAB measurements were taken at both the *file* level and the *function* level. In AVL measurements were taken at the *function* level only. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table D.3.

Metric	Pre-Value	Post-Value
RtnComNbr	0.0000	0.0000
RtnComVol	0.0000	0.0000
RtnCtlCplAvg	4.0483	4.0483
RtnCtlCplSum	64.7692	64.7692
RtnCtlCplMax	9.3003	9.3003
RtnExeCplAvg	7.2738	7.2738
RtnExeCplSum	308.3121	308.3121
RtnExeCplMax	18.2973	18.2973
RtnStmNbr	56.5222	56.5222
RtnXpdStmNbr	99.9482	99.9482
RtnCtlStmNbr	15.3669	15.3669
RtnIfNbr	8.0074	8.0074
RtnSwitchNbr	0.0030	0.0030
RtnLabelNbr	2.4630	2.4630
RtnCaseNbr	0.0266	0.0266
RtnDefaultNbr	0.0000	0.0000
RtnLopNbr	1.6938	1.6938
RtnReturnNbr	1.1036	1.1036
RtnGotoNbr	3.9571	3.9571
RtnContinueNbr	0.5888	0.5888
RtnBreakNbr	0.0133	0.0133
RtnExeStmNbr	32.6672	32.6672
RtnSysExitNbr	0.0000	0.0000
RtnDecStmNbr	8.4882	8.4882
RtnTypeDecNbr	0.0000	0.0000
RtnObjDecNbr	8.4867	8.4867

continued on next page

continued from previous page		
Metric	Pre-Value	Post-Value
RtnPrmNbr	2.1553	2.1553
RtnFctDecNbr	0.0015	0.0015
RtnStmNstLvlSum	1.1562	1.1562
RtnStmNstLvlAvg	104.1405	104.1405
RtnNstLvlMax	2.4734	2.4734
RtnScpNstLvlAvg	1.7935	1.7935
RtnScpNstLvlSum	29.7544	29.7544
RtnScpNbr	10.9660	10.9660
RtnXplCalNbr	22.0414	22.0414
RtnXplCastNbr	1.2589	1.2589
RtnCycCplNbr	10.7278	10.7278
OpdNbr	179.5680	179.5680
OpdUnqNbr	47.5769	47.5769
OprNbr	237.6716	237.6716
OprUnqNbr	14.8003	14.8033
HalDif	23.5483	23.5535
HalEff	202943.3935	202975.6943
HalLen	417.2396	417.2396
HalLvl	0.2173	0.2173
HalVoc	62.3772	62.3802
HalVol	3060.1143	3060.1412
RtnLnsNbr	72.4719	72.4719
RtnStxErrNbr	0.0000	0.0000
MI1	71.9263	71.9256
MI2	36.6910	36.6902
MI3	61.3768	61.3763

Table D.3: File level maintainability metrics on the WELTAB system before and after integer divide optimization

It is interesting to observe that most of the metrics did not change at all, and even those that did changed only slightly. These measures alone show that the new optimized system is almost as maintainable as the original one. However, we know that the new system is less maintainable because some divide instructions of the original system got replaced with shift instructions which are less intuitive.

The few metrics which increased slightly are the Halstead metrics OprUnqNbr, HalDif, HalEff, HalVoc, and HalVol. These metrics point out the fact that the new optimized code is slightly less maintainable than the original one. Thus, *Integer Divide Optimization* had as a result that maintainability was affected negatively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table D.4, and on the optimized version of AVL in Table D.5. All those measurements also show a decrease in maintainability after integer divide optimization.

Function	Metric	Pre-Value	Post-Value
wcre-showdone	MI1	70.05	69.90
	MI2	22.44	22.25
	MI3	48.00	47.88
weltab-showdone	MI1	70.05	69.91
	MI2	22.44	22.27
	MI3	48.00	47.89

Table D.4: Function level maintainability metrics on the WELTAB system before and after integer divide optimization

Function	Metric	Pre-Value	Post-Value
ubi_cacheGet	MI1	88.40	88.04
	MI2	87.16	86.71
	MI3	104.19	103.90

Table D.5: Function level maintainability metrics on the AVL system before and after integer divide optimization

D.2.3 Address Optimization

The objective of this performance optimization activity was to fit all the global scalar variables of WELTAB in a global variable pool. Then, each of the global scalar variables gets accessed via one pointer and an offset, instead of via constant address. This way, more expensive load and store sequences are avoided and code size is reduced. [16]

This is an example of how the global variables were declared and referenced in the original WELTAB system:

```
int nwrite;
int untspilt;
int untavcbs;
int untstart;
int untnprec;
int untwards;
int unitno;

void f (void)
{
  unitno = 10;
  return;
}
```

Below is the new code fragment after the global variables got mapped into a global memory pool. As we can see, the global variable *unitno* is now referenced by adding an offset 6 to the pointer *AddressOpt*.

```
int AddrOpt[7];
int *AddressOpt = &AddrOpt[0];

void f (void)
{
  *(AddressOpt+6) = 10;
  return;
}
```

This heuristic was implemented in WELTAB only, at both the *file* level and the *function* level. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table D.6.

Metric	Pre-Value	Post-Value
RtnComNbr	0.0000	0.0000
RtnComVol	0.0000	0.0000
RtnCtlCplAvg	4.0483	4.0552
RtnCtlCplSum	64.7692	65.0488
RtnCtlCplMax	9.3003	9.3846
RtnExeCplAvg	7.2738	7.2759
RtnExeCplSum	308.3121	308.7840
RtnExeCplMax	18.2973	18.3047
RtnStmNbr	56.5222	56.5222
RtnXpdStmNbr	99.9482	99.9482
RtnCtlStmNbr	15.3669	15.3669
RtnIfNbr	8.0074	8.0074
RtnSwitchNbr	0.0030	0.0030
RtnLabelNbr	2.4630	2.4630
RtnCaseNbr	0.0266	0.0266
RtnDefaultNbr	0.0000	0.0000
RtnLopNbr	1.6938	1.6938
RtnReturnNbr	1.1036	1.1036
RtnGotoNbr	3.9571	3.9571
RtnContinueNbr	0.5888	0.5888
RtnBreakNbr	0.0133	0.0133
RtnExeStmNbr	32.6672	32.6672
RtnSysExitNbr	0.0000	0.0000
RtnDecStmNbr	8.4882	8.4882
RtnTypeDecNbr	0.0000	0.0000
RtnObjDecNbr	8.4867	8.4867

continued on next page

continued from previous page		
Metric	Pre-Value	Post-Value
RtnPrmNbr	2.1553	2.1553
RtnFctDecNbr	0.0015	0.0015
RtnStmNstLvlSum	1.1562	1.1562
RtnStmNstLvlAvg	104.1405	104.1405
RtnNstLvlMax	2.4734	2.4734
RtnScpNstLvlAvg	1.7935	1.7935
RtnScpNstLvlSum	29.7544	29.7544
RtnScpNbr	10.9660	10.9660
RtnXplCalNbr	22.0414	22.0414
RtnXplCastNbr	1.2589	1.2589
RtnCycCplNbr	10.7278	10.7278
OpdNbr	179.5680	179.8772
OpdUnqNbr	47.5769	47.5784
OprNbr	237.6716	238.8210
OprUnqNbr	14.8003	14.8018
HalDif	23.5483	23.5686
HalEff	202943.3935	204257.1457
HalLen	417.2396	418.6982
HalLvl	0.2173	0.2173
HalVoc	62.3772	62.3802
HalVol	3060.1143	3071.5034
RtnLnsNbr	72.4719	72.4719
RtnStxErrNbr	0.0000	0.0000
MI1	71.9263	71.8982
MI2	36.6910	36.6559
MI3	61.3768	61.3547

Table D.6: File level maintainability metrics on the WELTAB system before and after address optimization

As we can see in this table, most measurements remained unchanged because of this optimization. The most significant changes appeared in the Halstead metrics.

All of the Halstead base metrics and derived metrics increased. These measures include HalVoc, HalVol, HalLvl, HalLen, HalEff, HalDif, OprUnqNbr, OprNbr, OpdUnqNbr, and OpdNbr. These increases can be attributed to the increases in the number of operators and operands. These increases resulted from the fact that in the new version of WELTAB global scalar variables get accessed by adding an offset to a pointer.

Another interesting result is the fact that RtnCtlCplAvg and RtnExeCplAvg also increased slightly. This implies that the total complexity of the decision statements, loop statements and executable statements increased. This increase can also be attributed to the increases in the number of operators and operands. These increases resulted from the fact that in the optimized version of WELTAB global scalar variables get accessed by adding an offset to a pointer.

The final observation we can make is that all the Maintainability Indexes (MIs) decreased. These descreases can be attributed to the fact that all Halstead's metrics (variables that affect the MIs) increased. Thus, *Address Optimization* had as a result that maintainability was affected negatively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table D.7. All those measurements also show a decrease in maintainability after address optimization.

Function	Metric	Pre-Value	Post-Value
cmprec-xfix	MI1	62.39	62.37
	MI2	-18.10	-18.13
	MI3	11.03	11.01
cmprec-prec	MI1	67.49	67.46
	MI2	11.60	11.55
	MI3	38.35	38.32
cmprec-vedt	MI1	62.29	62.26
	MI2	-18.78	-18.81
	MI3	10.39	10.37
cmprec-vset	MI1	75.88	75.89

continued on next page

continued from previous page			
Function	Metric	Pre-Value	Post-Value
	MI2	41.99	42.00
	MI3	64.84	64.84
cmprec-vfix	MI1	62.45	62.42
	MI2	-17.06	-17.09
	MI3	12.04	12.02
files-rsprtpag	MI1	65.23	65.22
	MI2	1.74	1.73
	MI3	29.54	29.54
files-prtpag	MI1	65.20	65.19
	MI2	1.62	1.60
	MI3	29.43	29.42
report-fixw	MI1	75.56	75.57
	MI2	40.88	40.89
	MI3	63.87	63.88
report-cmut	MI1	70.77	70.78
	MI2	21.93	21.93
	MI3	47.15	47.15
report-chead	MI1	81.41	81.41
	MI2	62.78	62.78
	MI3	83.05	83.05
report-rsum	MI1	68.48	68.48
	MI2	13.74	13.75
	MI3	40.03	40.03
report-lans	MI1	67.99	67.99
	MI2	11.23	11.23
	MI3	37.75	37.75
report-cnv1a	MI1	64.20	64.13
			continued on next page

continued from previous page			
Function	Metric	Pre-Value	Post-Value
	MI2	-10.32	-10.41
	MI3	17.96	17.91
report-canv	MI1	63.18	63.12
	MI2	-16.50	-16.58
	MI3	12.26	12.21
weltab-sped	MI1	68.32	68.25
	MI2	9.82	9.74
	MI3	36.19	36.14
weltab-poll	MI1	64.70	64.66
	MI2	-4.10	-4.15
	MI3	23.95	23.92
weltab-spol	MI1	63.64	63.60
	MI2	-10.60	-10.64
	MI3	17.94	17.91
weltab-getprec	MI1	79.08	78.63
	MI2	56.93	56.36
	MI3	78.29	77.93
weltab-pget	MI1	64.15	63.73
	MI2	-6.30	-6.82
	MI3	22.00	21.67
weltab-showpoll	MI1	67.49	67.36
	MI2	15.32	15.16
	MI3	42.07	41.97
weltab-showdone	MI1	70.05	69.91
	MI2	22.44	22.27
	MI3	48.00	47.89
weltab-	MI1	73.18	73.12
			continued on next page

continued from previous page			
Function	Metric	Pre-Value	Post-Value
allowcard	MI2	34.66	34.59
	MI3	58.77	58.72

Table D.7: Function level maintainability metrics on the WELTAB system before and after address optimization

D.2.4 Function Inlining

The objective of this performance optimization activity was to eliminate the overhead associated with calling and returning from a function, by expanding the body of the function inline.

For example, in the code fragment below, the function add() can be expanded inline at the call site in the function sub().

```
int add (int x, int y)
{
  return x + y;
}


int sub (int x, int y)
{
  return add (x, -y);
}
```

Expanding add() at the call site in sub() yields:

```
int sub (int x, int y)
{
  return x + -y;
}
```

Function inlining usually increases code space, which is affected by the size of the inlined function, and the number of call sites that are inlined.

This heuristic was implemented in both WELTAB and AVL. In WELTAB measurements were taken at both the *file* level and the *function* level. In AVL measurements were taken at the *function* level only. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table D.8.

Metric	Pre-Value	Post-Value
RtnComNbr	0.0000	0.0000
RtnComVol	0.0000	0.0000
RtnCtlCplAvg	4.0483	4.0593
RtnCtlCplSum	64.7692	63.7246
RtnCtlCplMax	9.3003	9.2083
RtnExeCplAvg	7.2738	7.0388
RtnExeCplSum	308.3121	324.3207
RtnExeCplMax	18.2973	17.6178
RtnStmNbr	56.5222	58.4692
RtnXpdStmNbr	99.9482	103.4348
RtnCtlStmNbr	15.3669	15.3315
RtnIfNbr	8.0074	8.1667
RtnSwitchNbr	0.0030	0.0018
RtnLabelNbr	2.4630	2.3533
RtnCaseNbr	0.0266	0.0163
RtnDefaultNbr	0.0000	0.0000
RtnLopNbr	1.6938	1.7409
RtnReturnNbr	1.1036	1.0851
RtnGotoNbr	3.9571	3.7609
RtnContinueNbr	0.5888	0.5616
RtnBreakNbr	0.0133	0.0145
RtnExeStmNbr	32.6672	34.5562
RtnSysExitNbr	0.0000	0.0000
RtnDecStmNbr	8.4882	8.5815
RtnTypeDecNbr	0.0000	0.0000
RtnObjDecNbr	8.4867	8.5815
RtnPrmNbr	2.1553	2.1667
RtnFctDecNbr	0.0015	0.0000

continued on next page

continued from previous page		
Metric	Pre-Value	Post-Value
RtnStmNstLvlSum	1.1562	1.1675
RtnStmNstLvlAvg	104.1405	108.2319
RtnNstLvlMax	2.4734	2.5272
RtnScpNstLvlAvg	1.7935	1.8177
RtnScpNstLvlSum	29.7544	31.0471
RtnScpNbr	10.9660	11.3388
RtnXplCalNbr	22.0414	22.6975
RtnXplCastNbr	1.2589	1.1540
RtnCycCplNbr	10.7278	10.9239
OpdNbr	179.5680	185.8877
OpdUnqNbr	47.5769	46.1667
OprNbr	237.6716	244.0833
OprUnqNbr	14.8003	14.6902
HalDif	23.5483	24.8862
HalEff	202943.3935	223962.9441
HalLen	417.2396	429.9710
HalLvl	0.2173	0.2226
HalVoc	62.3772	60.8569
HalVol	3060.1143	3149.6310
RtnLnsNbr	72.4719	76.3424
RtnStxErrNbr	0.0000	0.0000
MI1	71.9263	71.4982
MI2	36.6910	35.5612
MI3	61.3768	60.4460

Table D.8: File level maintainability metrics on the WELTAB system before and after Function Inlining

All of the Halstead base metrics and derived metrics increased. These measures include HalVol, HalLen, HalEff, HalDif, OprNbr, and OpdNbr. These increases can be attributed to the increases in the number of operators and operands. These increases resulted from the fact that in the new optimized version of WELTAB the amount of source code has increased.

Other measurements that increased are the number of statements in the routine (RtnStmNbr), and executable statements (RtnExeStmNbr). These increases can also be attributed to the increase in the amount of source code in the new version of WELTAB. Thus, it makes sense to conclude that *Function Inlining* had a negative effect on main memory utilization.

The final observation we can make is that all the Maintainability Indexes (MIs) decreased. These descreases can be attributed to the fact that all Halstead's metrics and lines of code (variables that affect the MIs) increased. Thus, *Function Inlining* had as a result that maintainability was affected negatively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table D.9, and on the optimized version of AVL in Table D.10. All those measurements also show a decrease in maintainability after function inlining.

Function	Metric	Pre-Value	Post-Value
weltab-poll	MI1	64.70	64.19
	MI2	-4.10	-4.33
	MI3	23.95	20.95
weltab-spol	MI1	63.64	63.21
	MI2	-10.60	-11.56
	MI3	17.94	15.18
report-cand	MI1	80.68	80.68
	MI2	56.09	56.09
	MI3	76.71	76.71
report.rsum	MI1	68.48	67.94
	MI2	13.74	12.00
	MI3	40.03	38.54
report-cnv1a	MI1	64.20	61.66
	MI2	-10.32	-11.30
	MI3	17.96	16.16
continued on next page			

continued from previous page			
Function	Metric	Pre-Value	Post-Value
report-canvw	MI1	77.14	75.11
	MI2	46.06	39.07
	MI3	68.32	62.27
report-dhead	MI1	78.83	73.16
	MI2	52.48	44.72
	MI3	73.96	68.83
report-canv	MI1	63.18	61.48
	MI2	-16.50	-17.20
	MI3	12.26	9.34
Baselib-setdate	MI1	88.86	71.99
	MI2	85.25	64.20
	MI3	102.06	72.86
Baselib-cvec	MI1	79.81	76.68
	MI2	56.15	48.85
	MI3	77.16	66.33

Table D.9: Function level maintainability metrics on the WELTAB system before and after function inlining

Function	Metric	Pre-Value	Post-Value
ubi_btInsert	MI1	77.85	77.73
	MI2	47.39	47.24
	MI3	69.32	69.22
ubi_cache Delete	MI1	91.18	90.59
	MI2	94.48	93.76
	MI3	110.22	109.76
ubi_cache	MI1	91.96	91.32
			continued on next page

continued from previous page			
Function	Metric	Pre-Value	Post-Value
Reduce	MI2	93.33	92.53
	MI3	108.70	108.19
ubi_cacheSet MaxEntries	MI1	92.79	87.15
	MI2	101.13	88.93
	MI3	116.14	106.58
ubi_cacheSet MaxMemory	MI1	92.79	87.15
	MI2	101.16	88.98
	MI3	116.14	106.58
ubi_cachePut	MI1	91.44	84.88
	MI2	91.20	79.57
	MI3	106.81	98.23

Table D.10: Function level maintainability metrics on the AVL system before and after function inlining

D.2.5 Elimination of GOTO statements

The objective of this maintenance optimization activity was to minimize the number of GOTO statements in WELTAB. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.

It is important to note that the original WELTAB C++ source code contained a very large number of GOTO statements. It was not possible to eliminate all GOTO statements, since in many cases removing them would have altered the source code's control flow. Each GOTO statement that was eliminated got replaced with a block of executable statements, ending with a return statement. Thus, it was ensured that the control flow in the optimized version was exactly the same as in the original version of WELTAB.

This heuristic was implemented in WELTAB only. Measurements were taken at both the *file* level and the *function* level. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table D.11.

Metric	Pre-Value	Post-Value
RtnComNbr	0.0000	0.0000
RtnComVol	0.0000	0.0000
RtnCtlCplAvg	4.0483	4.2050
RtnCtlCplSum	64.7692	68.3996
RtnCtlCplMax	9.3003	9.7183
RtnExeCplAvg	7.2738	7.5615
RtnExeCplSum	308.3121	324.2096
RtnExeCplMax	18.2973	19.4574
RtnStmNbr	56.5222	59.0928
RtnXpdStmNbr	99.9482	102.9749
RtnCtlStmNbr	15.3669	16.0786
RtnIfNbr	8.0074	8.4847
RtnSwitchNbr	0.0030	0.0055
RtnLabelNbr	2.4630	2.3504
RtnCaseNbr	0.0266	0.0491
RtnDefaultNbr	0.0000	0.0000
RtnLopNbr	1.6938	1.7085
RtnReturnNbr	1.1036	1.6725
RtnGotoNbr	3.9571	3.5917
RtnContinueNbr	0.5888	0.6026
RtnBreakNbr	0.0133	0.0131
RtnExeStmNbr	32.6672	34.3941
RtnSysExitNbr	0.0000	0.0000
RtnDecStmNbr	8.4882	8.6201
RtnTypeDecNbr	0.0000	0.0000
RtnObjDecNbr	8.4867	8.6179

continued on next page

continued from previous page		
Metric	Pre-Value	Post-Value
RtnPrmNbr	2.1553	2.1987
RtnFctDecNbr	0.0015	0.0022
RtnStmNstLvlSum	1.1562	1.1771
RtnStmNstLvlAvg	104.1405	109.3231
RtnNstLvlMax	2.4734	2.5240
RtnScpNstLvlAvg	1.7935	1.8219
RtnScpNstLvlSum	29.7544	31.1114
RtnScpNbr	10.9660	11.4825
RtnXplCalNbr	22.0414	23.6736
RtnXplCastNbr	1.2589	1.3930
RtnCycCplNbr	10.7278	11.2424
OpdNbr	179.5680	187.6321
OpdUnqNbr	47.5769	49.0338
OprNbr	237.6716	250.5524
OprUnqNbr	14.8003	15.3788
HalDif	23.5483	24.8515
HalEff	202943.3935	218348.6089
HalLen	417.2396	438.1845
HalLvl	0.2173	0.1957
HalVoc	62.3772	64.4127
HalVol	3060.1143	3209.2513
RtnLnsNbr	72.4719	75.9356
RtnStxErrNbr	0.0000	0.0000
MI1	71.9263	71.6085
MI2	36.6910	35.4542
MI3	61.3768	60.2877

Table D.11: File level maintainability metrics on the WELTAB system before and after eliminating GOTO statements

It is important to note that maintainability did get improved by eliminating GOTO statements. Elimination of GOTO statements is the only way to minimize the number of unconditional branches in source code. Decreasing the number of unconditional branches is a key factor in improving maintainability, as it can assist a maintainer in understanding the source code of a system. [6] In our measurements, the number of unconditional branches is shown by the metric RtnGotoNbr, which decreased significantly after GOTO statements were eliminated.

However, elimination of GOTO statements also affects other characteristics of source code in varying ways, and thus maintainability may get affected in different ways. After eliminating GOTO statements many of the DATRIX measurements showed that source code became slightly less maintainable. These measurements are shown in Table D.11.

Eliminating GOTO statements had as a consequence that the source code's complexity increased. This is shown by the fact that all measurements related to source code complexity went up. These measures include RtnCtlCplAvg, RtnExeCplAvg, and RtnCycCplNbr. These changes can easily be attributed to the fact that each GOTO statement in the C++ version of WELTAB got replaced with blocks of executable source code.

Other measurements that increased are the number of statements in the routine (RtnStmNbr), the number of control flow statements (RtnCtlStmNbr), executable statements (RtnExeStmNbr), declarative statements (RtnDecStmNbr), variable/object decalaration statements (RtnObjStmNbr), the number of function/method calls (RtnFctDecNbr) and the number of return statements (RtnReturnNbr). These increases can also be attributed to the blocks of executable source code which have replaced the GOTO statements in the new version of WELTAB. Thus, it makes sense to conclude that elimination of GOTO statements had a negative effect on main memory utilization.

Another interesting result is the fact that all of the Halstead base metrics and derived metrics increased as well. These measures include HalVoc, HalVol, HalLvl, HalLen, HalEff, HalDif, OprUnqNbr, OprNbr, OpdUnqNbr, and OpdNbr. These increases can be attributed to the increase in the number of operators and operands, which resulted from the blocks of executable source code which replaced the GOTO statements.

The final observation we can make from the metrics is that all the Maintainability Indexes (MIs) decreased. These descreases can be attributed to the fact that all Halstead's metrics, McCabe's Cyclomatic Complexity and lines of code (variables that affect the MIs) increased.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table D.12. All those measurements also show a decrease in maintainability after eliminating GOTO statements.

Function	Metric	Pre-Value	Post-Value
weltab-sped	MI1	68.32	67.44
	MI2	9.82	5.22
	MI3	36.19	31.99
weltab-poll	MI1	63.64	63.87
	MI2	-10.60	-6.72
	MI3	17.94	21.72
weltab-spol	MI1	63.64	62.85
	MI2	-10.60	-13.07
	MI3	17.94	15.83
weltab-allowcard	MI1	73.18	72.83
	MI2	34.66	33.70
	MI3	58.77	57.96
cmprec-xfix	MI1	62.45	62.04
	MI2	-17.06	-19.00
	MI3	12.04	10.28
cmprec-vfix	MI1	62.45	62.09
	MI2	-17.06	-18.01
	MI3	12.04	11.24
cmprec-vset	MI1	75.88	75.11
	MI2	41.99	39.24
	MI3	64.84	62.45
cmprec-vedt	MI1	62.29	61.94
	MI2	-18.78	-19.72
	MI3	10.39	9.61
cmprec-prec	MI1	67.49	67.36
	MI2	11.60	10.81

continued on next page

continued from previous page			
Function	Metric	Pre-Value	Post-Value
	MI3	38.35	37.62
report-cnv1a	MI1	64.20	63.96
	MI2	-10.32	-10.72
	MI3	17.96	17.67
report-cmut	MI1	70.77	70.62
	MI2	21.93	21.46
	MI3	47.15	46.75
report-fixw	MI1	75.56	74.94
	MI2	40.88	39.25
	MI3	63.87	62.53

Table D.12: Function level maintainability metrics on the WELTAB system before and after eliminating GOTO statements

D.2.6 Dead Code Elimination

The objective of this maintenance optimization activity was to eliminate dead code that was unreachable or that did not affect the program. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.

It is important to note that the original WELTAB C++ source code contained a large amount of dead code. It cannot be certain that all dead code was eliminated. However, after dead code was eliminated on some source files, the size of the files decreased by almost half their original size. This fact alone points out the importance of dead code elimination, not only for maintainability purposes, but also for space performance purposes.

This heuristic was implemented in WELTAB only, at both the *file* level and the *function* level. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table D.13.

Metric	Pre-Value	Post-Value
RtnComNbr	0.0000	0.0000
RtnComVol	0.0000	0.0000
RtnCtlCplAvg	4.0483	3.9224
RtnCtlCplSum	64.7692	26.7616
RtnCtlCplMax	9.3003	6.8142
RtnExeCplAvg	7.2738	6.9737
RtnExeCplSum	308.3121	132.3684
RtnExeCplMax	18.2973	13.7926
RtnStmNbr	56.5222	26.8576
RtnXpdStmNbr	99.9482	39.2848
RtnCtlStmNbr	15.3669	6.9195
RtnIfNbr	8.0074	3.5077
RtnSwitchNbr	0.0030	0.0031
RtnLabelNbr	2.4630	0.6533
RtnCaseNbr	0.0266	0.0279
RtnDefaultNbr	0.0000	0.0000
RtnLopNbr	1.6938	0.8669
RtnReturnNbr	1.1036	1.4149
RtnGotoNbr	3.9571	0.9009
RtnContinueNbr	0.5888	0.2043
RtnBreakNbr	0.0133	0.0217
RtnExeStmNbr	32.6672	14.4272
RtnSysExitNbr	0.0000	0.0000
RtnDecStmNbr	8.4882	5.5108
RtnTypeDecNbr	0.0000	0.0000
RtnObjDecNbr	8.4867	5.5108

continued on next page

continued from previous page		
Metric	Pre-Value	Post-Value
RtnPrmNbr	2.1553	2.6780
RtnFctDecNbr	0.0015	0.0000
RtnStmNstLvlSum	1.1562	1.0598
RtnStmNstLvlAvg	104.1405	40.6842
RtnNstLvlMax	2.4734	2.2043
RtnScpNstLvlAvg	1.7935	1.6468
RtnScpNstLvlSum	29.7544	13.1610
RtnScpNbr	10.9660	5.6254
RtnXplCalNbr	22.0414	9.3932
RtnXplCastNbr	1.2589	0.3406
RtnCycCplNbr	10.7278	5.4025
OpdNbr	179.5680	77.7245
OpdUnqNbr	47.5769	25.4861
OprNbr	237.6716	103.7245
OprUnqNbr	14.8003	12.4520
HalDif	23.5483	16.4142
HalEff	202943.3935	59274.8497
HalLen	417.2396	181.4489
HalLvl	0.2173	0.1830
HalVoc	62.3772	37.9381
HalVol	3060.1143	1198.8055
RtnLnsNbr	72.4719	35.3560
RtnStxErrNbr	0.0000	0.0000
MI1	71.9263	77.2713
MI2	36.6910	56.6653
MI3	61.3768	78.8650

Table D.13: File level maintainability metrics on the WELTAB system before and after eliminating dead code

Eliminating dead code had as a consequence that the source code's complexity decreased. This is shown by the fact that all metrics related to source code complexity went down, such as RtnCtlCplAvg, RtnExeCplAvg, and RtnCycCplNbr. These decreases can be attributed to the blocks of executable source code eliminated in the new optimized system.

Eliminating dead code had as a consequence that all of the Halstead base metrics and derived metrics decreased. These measures include HalVoc, HalVol, HalLvl, HalLen, HalEff, HalDif, OprUnqNbr, OprNbr, OpdUnqNbr, and OpdNbr. These decreases can be attributed to the decrease in the number of operators and operands, which resulted from the blocks of executable source code eliminated in the new optimized system.

All the Maintainability Indexes (MIs) increased significantly, by nearly 30to the fact that all Halstead's metrics (variables that affect the MIs) decreased. Thus, *Dead Code Elimination* had as a result that maintainability was affected positively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table D.14. All those measurements also show an increase in maintainability after eliminating dead code.

Function	Metric	Pre-Value	Post-Value
report	MI1	70.43	76.32
	MI2	36.22	55.32
	MI3	61.43	73.67
card	MI1	72.76	73.23
	MI2	38.32	49.23
	MI3	62.78	71.06
weltab	MI1	70.23	75.98
	MI2	39.03	49.32
	MI3	61.43	77.32
files	MI1	69.45	74.32
	MI2	40.01	56.98
	MI3	62.67	78.02
cmprec	MI1	68.04	72.76

continued on next page

continued from previous page

Function	Metric	Pre-Value	Post-Value
	MI2	36.43	51.56
	MI3	64.98	77.32

Table D.14: Function level maintainability metrics on the WELTAB system before and after eliminating dead code

D.2.7 Elimination of Global Data Types and Data Structures

The objective of this maintenance optimization activity was to turn global data types and data structures to local. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.

This heuristic was implemented in WELTAB only, and measurements were taken at both the *file* level and the *function* level. The *file* level measurements taken on the new optimized version of WELTAB are shown in Table D.15.

Metric	Pre-Value	Post-Value
RtnComNbr	0.0000	0.0000
RtnComVol	0.0000	0.0000
RtnCtlCplAvg	4.0483	4.0364
RtnCtlCplSum	64.7692	64.5782
RtnCtlCplMax	9.3003	9.2729
RtnExeCplAvg	7.2738	7.2523
RtnExeCplSum	308.3121	307.4027
RtnExeCplMax	18.2973	18.2434
RtnStmNbr	56.5222	56.3555
RtnXpdStmNbr	99.9482	99.6534
RtnCtlStmNbr	15.3669	15.3215

continued on next page

continued from previous page		
Metric	Pre-Value	Post-Value
RtnIfNbr	8.0074	7.9838
RtnSwitchNbr	0.0030	0.0029
RtnLabelNbr	2.4630	2.4558
RtnCaseNbr	0.0266	0.0265
RtnDefaultNbr	0.0000	0.0000
RtnLopNbr	1.6938	1.6888
RtnReturnNbr	1.1036	1.1003
RtnGotoNbr	3.9571	3.9454
RtnContinueNbr	0.5888	0.5870
RtnBreakNbr	0.0133	0.0133
RtnExeStmNbr	32.6672	32.5708
RtnSysExitNbr	0.0000	0.0000
RtnDecStmNbr	8.4882	8.4631
RtnTypeDecNbr	0.0000	0.0000
RtnObjDecNbr	8.4867	8.4617
RtnPrmNbr	2.1553	2.1490
RtnFctDecNbr	0.0015	0.0015
RtnStmNstLvlSum	1.1562	1.1528
RtnStmNstLvlAvg	104.1405	103.8333
RtnNstLvlMax	2.4734	2.4690
RtnScpNstLvlAvg	1.7935	1.7912
RtnScpNstLvlSum	29.7544	29.6696
RtnScpNbr	10.9660	10.9366
RtnXplCalNbr	22.0414	21.9764
RtnXplCastNbr	1.2589	1.2552
RtnCycCplNbr	10.7278	10.6991
OpdNbr	179.5680	179.0383
		continued on next page

continued from previous page		
Metric	Pre-Value	Post-Value
OpdUnqNbr	47.5769	47.4366
OprNbr	237.6716	236.9705
OprUnqNbr	14.8003	14.7566
HalDif	23.5483	23.4759
HalEff	202943.3935	202344.7375
HalLen	417.2396	416.0088
HalLvl	0.2173	0.2137
HalVoc	62.3772	62.1932
HalVol	3060.1143	3051.0844
RtnLnsNbr	72.4719	72.2611
RtnStxErrNbr	0.0000	0.0000
MI1	71.9263	71.9391
MI2	36.6910	36.7616
MI3	61.3768	61.4414

Table D.15: File level maintainability metrics on the WELTAB system before and after eliminating global data types and data structures

Eliminating global data structures had as a consequence that all of the Halstead base metrics and derived metrics decreased. These measures include HalVoc, HalVol, HalLvl, HalLen, HalEff, HalDif, OprUnqNbr, OprNbr, OpdUnqNbr, and OpdNbr.

The source code's complexity also decreased. This is shown by the fact that all metrics related to source code complexity went down. These metrics include RtnCtlCplAvg, RtnExeCplAvg, and RtnCycCplNbr.

The final observation we can make is that all the Maintainability Indexes (MIs) increased. These increases can be attributed to the fact that all Halstead's metrics (variables that affect the MIs) decreased. Thus, *Elimination of Global Data Types and Data Structures* had as a result that maintainability was affected positively in the optimized system.

The *function* level measurements taken on the new optimized version of WELTAB are shown in Table D.16. All those measurements also show an increase in maintainability after eliminating global data types and data structures.

Function	Metric	Pre-Value	Post-Value
report	MI1	71.92	81.02
	MI2	36.69	38.91
	MI3	61.38	62.04
weltab	MI1	73.18	74.56
	MI2	38.55	39.76
	MI3	65.44	65.59

Table D.16: Function level maintainability metrics on the WELTAB system before and after eliminating global data types and data structures

D.2.8 Maximization of Cohesion

The objective of this maintenance optimization activity was to split a class with low cohesion into many smaller classes, each of which has higher cohesion. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.

This heuristic was implemented in AVL only, and measurements were taken at the *function* level only. The *function* level measurements taken on the new optimized version of AVL are shown in Table D.17. All those measurements show an increase in maintainability after maximizing cohesion.

Function	Metric	Pre-Value	Post-Value
SampleRec	MI1	93.65	94.66
	MI2	103.03	105.01
	MI3	119.21	121.89

Table D.17: Function level maintainability metrics on the AVL system before and after maximizing cohesion

D.2.9 Minimization of Coupling Through ADTs

The objective of this maintenance optimization activity was to eliminate variables declared within a class, which have a type of ADT that is another class definition. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.

This heuristic was implemented in AVL only, and measurements were taken at the *function* level only. The *function* level measurements taken on the new optimized version of AVL are shown in Table D.18. All those measurements show an increase in maintainability after minimizing coupling through ADTs.

Function	Metric	Pre-Value	Post-Value
ubi_cacheRoot	MI1	76.86	79.31
	MI2	98.77	102.67
	MI3	108.44	111.45
ubi_idbDB	MI1	83.46	85.18
	MI2	88.67	93.63
	MI3	99.46	106.32
ubi_btNode	MI1	92.76	96.17
	MI2	92.49	93.25
	MI3	116.21	117.38
ubi_idb FuncRec	MI1	81.07	88.93
	MI2	107.33	117.43
	MI3	127.32	139.87

Table D.18: Function level maintainability metrics on the AVL system before and after minimizing coupling through ADTs