Phase-ordering in optimizing compilers

Matthieu Quéva

Kongens Lyngby 2007
IMM-MSC-2007-71

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

The "quality" of code generated by compilers largely depends on the analyses and optimizations applied to the code during the compilation process. While modern compilers could choose from a plethora of optimizations and analyses, in current compilers the order of these pairs of analyses/transformations is fixed once and for all by the compiler developer. Of course there exist some flags that allow a marginal control of what is executed and how, but the most important source of information regarding what analyses/optimizations to run is ignored-the source code. Indeed, some optimizations might be better to be applied on some source code, while others would be preferable on another.

A new compilation model is developed in this thesis by implementing a Phase Manager. This Phase Manager has a set of analyses/transformations available, and can rank the different possible optimizations according to the current state of the intermediate representation. Based on this ranking, the Phase Manager can decide which phase should be run next.

Such a Phase Manager has been implemented for a compiler for a simple imperative language, the WHILE language, which includes several Data-Flow analyses. The new approach consists in calculating coefficients, called *metrics*, after each optimization phase. These metrics are used to evaluate where the transformations will be applicable, and are used by the Phase Manager to rank the phases.

The metrics are calculated using a new algorithm that approximates the data-flow analyses for the WHILE language. This algorithm skims through the intermediate representation of the program only once, and thus solves the analyses' equations faster than classical worklist algorithms.

In order to evaluate the metric-based approach, a benchmark suite is created. This suite generates a large amount of regular expressions that express various orders of optimizations. The metric-based phase-ordering is applied on several benchmark programs and compared to the best regular expression on each of these programs.

Finally, this thesis considers the interactions between the transformations considered, as well as the effects of these transformations on the size and speed of the program to improve the metric-based algorithm. The resulting versions of the phase-ordering mechanism are then evaluated using the benchmark suite.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Master of Science degree in engineering.

The thesis deals with the ordering of optimization phases in optimizing compilers. The main focus is on the application of a new approach based on the calculation of metrics by a Phase Manager to rank the optimizations. In particular, this Phase Manager has been implemented on top of a compiler written in Java for several Data-Flow analyses.

Lyngby, July 2007

Matthieu Quéva

Acknowledgements

I wish to thank my supervisor, Christian Probst, for his strong interest in my project and for his constructive criticisms during the period. We had a lot of interesting talks about this project and it was very nice to have a supervisor who followed closely my work, especially in moments of doubts and hesitations.

I would also like to thank the whole LBT (Language-Based Technology) group at IMM in which I worked during this thesis.

I finally would like to thank all my friends in Denmark that supported me during this thesis, including my girlfriend Julie who I should thank more often for everything.

Contents

Chapter 1

Introduction

Computer systems are used everywhere. From the desktop PC to the washing machine, as you take your car that uses a plethora of micro-controllers, from the high performance supercomputers to the tiny devices in mobile phones. A computer systems in itself is a combination of hardware and software. To control the hardware, programmers implement a series of instructions in a specific language called a programming language. The world depends on these programming languages, because each software in any computer system is written in some programming language. But before the machine can understand this language, it must be transformed into another low-level language executable by the computer.

The software that is responsible for this translation is called a *compiler*. The compilers are also used to optimize the input code in order to improve the performance of the application that has to be executed. In this thesis, we consider the issue of the phase ordering in the optimizing compilers, i.e trying to find the best order in which the different optimizations can be applied within the compiler to produce the most efficient code.

This preliminary chapter briefly introduces the different features of a compiler, from its structure to the optimizations and the issue of phase ordering. Then the thesis outline is considered.

1.1 Compiler and optimizations

We mentioned earlier the software programs called compilers, and in particular the optimizing compilers. This section deals with a description of what a compiler is and does, and what *optimizing compiler* really means.

1.1.1 Description of a compiler

A formal description of what is a compiler can be obtained from Aho et al. ([1]):

> *a* compiler *is a program that can read a program in one language - the* source *language - and translate it into an equivalent program in another language - the* target *language.*

The first programs to do this kind of translation were called *assemblers*, and have been available since the 1950s. They were just taking assembly code (e.g. "`addcc %r1,%r2,%r4`") as input and translating it into machine code (e.g "`0x88804002`"). Now, some of the current compilers can optimize the code in order to improve the performance of the resulting application, according to different goals (most often speed of execution, code size or power consumption).

These compilers are then called *optimizing* compilers, though the idea of an *optimal* program is sometimes ambiguous in these compilers. Indeed, the notion of optimality cannot exist in practice, because there are loops with conditional jumps for which no semantically equivalent time-optimal program exists on parallel machines ([22]). The structure of a typical multi-language, multi-target compiler is shown in Figure 1.1.

[1] defines the two parts in a traditional compiler:

- The *analysis* part that breaks up the program into pieces that are then used to create an intermediate representation. It can detect if the source program is syntactically wrong and report it to the user. In an ideal compiler, this part, also called the *frontend* of the compiler, is independent from the target language (and thus the target machine).
- The *synthesis* part that constructs the target program from the intermediate representation. This part, also called the *backend*, is ideally independent from the source program.

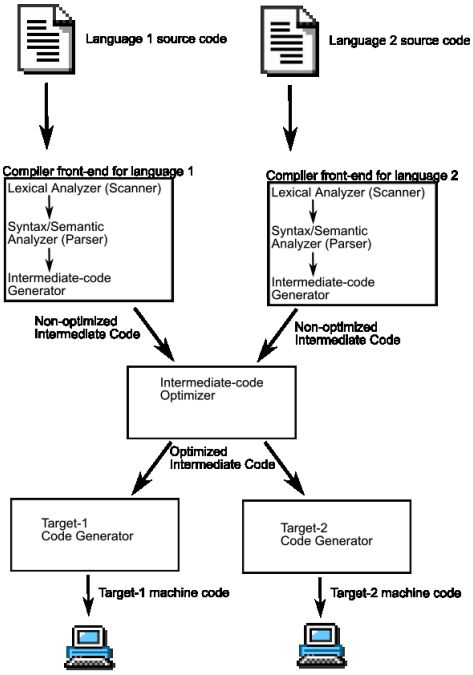


Figure 1.1: A multi-language, multi-target compiler ([5])

On the optimizing compilers, another part is included between these two parts: it is the machine- and language-independent optimization part that applies various optimizations on the intermediate representation, aiming at improving the target program's performance.

1.1.2 Optimizations

It can be very profitable to improve the object code produced by a compiler. Nowadays, a lot of applications have performance constraints: whenever a vendor advertises his new application, he is likely to emphasize its performance in order to attract the customers. Such constraints are also very frequent whenever the computer system is an embedded device. Indeed, these embedded systems face speed constraints, as many of the real-time systems; code size constraints, because the memory available is limited; and power consumption constraints, as power supply is also limited.

Thus, a lot of different optimizations are available in optimizing compilers and

can be applied on the intermediate code, as illustrated in Figure 1.1. Again, Muchnick ([15]) emphasizes the fact that "*optimization* is a misnomer - only very rarely does applying optimizations to a program result in object code whose performance is optimal, by any measure".

A simplified model of an optimization phase is:

$$Optimization = Analysis + Transformation$$

The analysis part is used to gather information about different parameters in the program, e.g. variables, instructions, structure of the program.... The transformation is the part that will use the results from the analysis and apply the changes on the program.

It is very profitable to apply optimizations on the intermediate representation of the program because:

1. The intermediate code representation is normalized and source-independent: thus programs written in different source languages can produce the similar intermediate code.
2. The intermediate code representation is also target-independent, so the optimizations can be applied on the same intermediate representation and then the target code can be generated for different target architectures.
3. The intermediate representation is semantically simpler than the source program, and is often represented as a parse tree, which simplify the analyses.

1.1.3 Phase ordering

The phase ordering problem has long been known to be a difficult dilemma for compiler writers ([25]). It consists on answering the question:

> *"Which number of optimizations should be applied to the program, and in which order, to achieve the greatest benefit?"*

One specific sequence of optimizations is highly unlikely to be the most effective sequence for every program on a given machine. Current optimizing compilers contain command-line flags (such as -O, -O2, -O3 for gcc [18]) that provide

the user with some pre-defined optimizations sequences. These sequences are worked out to be efficient optimizations orders, but these orders are still fixed for all programs. Then, they cannot be optimal for every program, especially because of the interactions between the different optimizations, and also because a program may require a certain order that will not be very efficient on another one.

1.2 Thesis outline

In this thesis, the issue of the phase ordering in the optimizing compilers will be addressed. The thesis will focus on several intra-procedural analyses and transformations, using one of the main approach to program analysis, namely Data Flow Analysis.

The next chapter deals with the theoretical background behind the phase-ordering problem. It addresses the previous work concerning this problem, as well as the data flow analyses and the different algorithms currently available to solve the equations derived from these analyses.

Chapter 3 describes the environment in which the phase-ordering problem has been addressed in this thesis. It describes the WHILE language and the analyses and transformations available in the WHILE compiler.

Chapter 4 introduces the new compilation model where another module, called the Phase Manager, has been added to the existing compiler model. It also covers the first approach to the phase-ordering problem, where regular expressions are used to express the order of optimization. Finally a benchmark suite is designed.

Chapter 5 covers the main approach to the problem considered in this thesis, where the Phase Manager uses coefficients, called *metrics*, to evaluate the effects of the different optimizations according to the state of the intermediate representation.

Chapter 6 establishes an experimental comparison between the best regular expressions from the benchmark suite and the metric-based compilation approach. It then considers the interactions between the different optimizations to try to improve the metrics' computation time.

Chapter 7 covers a small study on the effects of the optimizations used and an evolution of the phase-ordering algorithm toward a goal-dependent mecha-

nism.

Chapter 8 covers the design and the implementation of the Phase Manager in Java, and the different issues that occurred during this implementation.

Finally, Chapter 9 discusses several issues and perspectives of future work, and Chapter 10 presents the conclusion of the work done in this thesis.

Chapter 2

Theoretical background

This chapter first gives an overview of the previous work that has been attempted in order to find viable solutions to the phase-ordering problem in compilers. Then the second part of this chapter is dedicated to some basic knowledge about Data Flow Analysis, and the different frameworks and algorithms used to model the analyses and solve the different resulting equations. This chapter aims at giving the reader some knowledge about the different approaches of the research community concerning this major issue in compilers, as well as the necessary background for the different optimization techniques addressed in the rest of this thesis.

2.1 The Phase-Ordering Problem

Finding the best order in optimizing compilation is an old problem [22]. Most optimizing compilers contain tens of different optimization phases, but because of their interactions, the target code produced can differ depending on the order in which these optimization phases were applied.

Although the capabilities of optimizing compilers have grown over time, some researchers have been investigating the benefits of this work. In [20], Scott

describes (and evaluates) a pretty pessimistic assertion known as Proebsting's Law. This assertion is to be associated with the well-known Moore's Law ([14]), which states that the processors have been doubling in capabilities every 18 months, while according to Proebsting the capabilities of the optimizing compilers have been doubling every 18 *years*! After evaluating Proebsting's Law, Scott concludes on the relative veracity of Proebsting, but clearly emphasizes that the benefit from compiler research is still not negligible, as the developed compiler technology has been able to achieve up to an average speedup of 8.1.

2.1.1 Theoretical models

In order to have a better overview of the phase ordering problem, some researches have been attempted on a theoretical approach to this problem. In [22], Touati and Barthou provides a formalism for two different known problems, including the phase-ordering issue itself. This part relates their work on the modeling of the problem and their results concerning whether the phase-ordering problem can be decidable or not.

2.1.1.1 Formulation of the problem

In order to study the decidability of the phase-ordering problem, Touati and Barthou made a theoretical model in [22]. Starting from a finite set of optimization modules $\mathcal{M}$, the aim is to construct an algorithm $\mathcal{A}$ with four inputs: a performance evaluation function t, a program $\mathcal{P}$, an input data I and a desired execution time T. Then, this algorithm $\mathcal{A}$ must compute a finite sequence $s = m_n \circ m_{n-1} \circ ... \circ m_0, m_i \in \mathcal{M}^*$ that solves the following problem:

PB. 1 *Let $\mathcal{M}$ be a finite set of program transformations. For any performance evaluation function t, $\forall T \in \mathbb{N}$ an execution time (in processor clock cycles), $\forall \mathcal{P}$ a program, $\forall I$ input data, does there exist a sequence $s \in \mathcal{M}^*$ such that $t(s(\mathcal{P}), I) < T$? In other words, if we define the set:*

$$S_{\mathcal{M}}(t, \mathcal{P}, I, T) = \{s \in \mathcal{M}^* | t(s(\mathcal{P}), I) < T\}$$

is the set $S_{\mathcal{M}}(t, \mathcal{P}, I, T)$ empty ?

The decidability of this problem is very much linked to the different parameters in stake. Indeed, when making the assumption that there exists a program that can be optimized into an infinite number of different programs, they proved

that the problem is *undecidable*. This means that the number of different optimization sequences is infinite, but also that it is not possible to find a fixed point where all the remaining sequences create programs that have already been generated previously.

2.1.1.2 Simplification of the problem

In order to simplify this problem towards a decidable model, Touati and Barthou ([22]) consider two cases:

1. a model with compilation costs
2. the case of generative compilers

The compilation cost model introduces a function c that models a cost in the compilation. This cost can be the compilation time, the number of generated programs, the number of compilation sequences, etc... Adding this function makes the problem much easier, as it is a strictly increasing function, thus the algorithm $\mathcal{A}$ can trivially search for all the compilation sequences with bounded cost, and get the best one among them. This approach is often used in the iterative compilation (see Section 2.1.2).

The other approach they consider is the one-pass generative compilers. In these compilers, the intermediate program is optimized and generated in a one pass traversal of the parse tree. Whenever a part of the program is optimized by a set of compilation phases, the final code for this part is generated, so it is not possible for another optimization module to re-optimized this part. This approach aims at producing local optimized code instead of globally optimizing the program. [22] contains a simple algorithm to show that this instance of the phase ordering problem is decidable too. A synthesis of the decidability analysis of the phase ordering problem can be seen in Figure 2.1.

This theoretical approach by Touati and Barthou is an interesting step to model and understand the phase ordering problem, in order to come with efficient solutions. However, this model is still vague about the decidability of the problem when an evaluation function that does not require the execution of the program is used. Indeed, they considered that the evaluation function t is not an approximation, and thus, as their model does not take the underlying hardware of the target machine into account in the performance evaluation, only the real execution time can yet satisfy this condition.

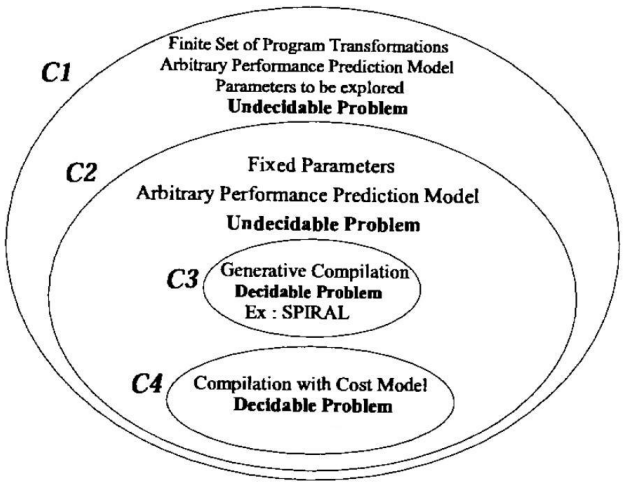


Figure 2.1: Classes of Phase-Ordering Problems [22]. The SPIRAL project is an example of a generative compiler [19]

This model however shows great hope for compiler's developers as the simplification of the phase-ordering problem can lead to make the problem decidable. The use of static performance evaluation techniques or the limitation of some parameters as the sequence length permits to obtain efficient optimized code, as can be seen in the next section.

2.1.2 Current compilation techniques

Currently, several types of compilation are used in an attempt to find optimal code. Some are requiring a longer compilation time than others, though most often producing better results. The two types of compilation techniques considered in this section are the profit-driven compilation and iterative compilation, with an emphasis on the latter one.

2.1.2.1 Profit-driven compilation

Profit-driven compilation consists in using static cost models in order to find the best order of optimization associated with a specific performance goal. The main issue with this type of compilation is that it heavily relies on the cost mod-

els, which are not reliable for more and more complex and dynamic architectures.

In [26], Zhao et al. evaluate the different optimization sequences to apply to a given program using a specific performance model. This model is determining the profit of the different optimization sequences according to three types of analytic models: code, optimization and resource models, considering resources like cache behaviour, registers and functional units.

The code model is a representation of the input code automatically generated by the optimizer. It expresses different characteristics of the code segment that can be targeted by an optimization or that can have an impact on one of the resources.
The optimization model is representing the impacts of a transformation on the resources and on the target code. This model is defined by the compiler writer whenever he is introducing a new optimization in the compiler.
The resource model consists in describing a specific resource according to the benefits and costs information in using this particular resource. This model is machine-dependent and must be modified whenever another target platform is used.

These three models provide information to a profitability engine which will compute the profit of the optimizations applied. In this particular model, and contrary to the theoretical model defined in [22], the optimizations does not consider the input data, and the same sequence of optimizations is applied whatever the values of the input. However, the experimental results in [26] show that it can be possible to find efficient optimization sequences without having to run the resulting code, though an accurate modeling of the resources, which has an important impact, can still be difficult for complex target architectures.

2.1.2.2 Iterative compilation

Another interesting approach is the field of iterative compilation. In this approach, the program is compiled multiple times iteratively, and at each iteration, a new optimization sequence is used, until an “optimal” solution is found. However, it is clear that trying all the optimization sequences possible is far from reasonable, as the optimization search space is much too wide to exhaustively enumerate all the possible sequences. In fact, the problem can be very complex as many optimizations are successful multiple times, making it impossible to fix the sequence length for all functions. However, some researchers have created efficient compilers with this iterative method, using several different methods to prune the search space, and fixing some of the parameters.

Three different methods are described in the following paragraphs: the Optimization-Space Exploration, the characterization of the optimization search space, and the use of heuristics.

The Optimization-Space Exploration approach

In [23], Triantafyllis et al. present an novel iterative compilation approach, called Optimization-Space Exploration (OSE). This method uses the classic iterative compilation techniques, optimizing each code segment with various optimizations configurations and evaluating the resulting code after optimization to find the best sequence, as can be seen in Figure 2.2.

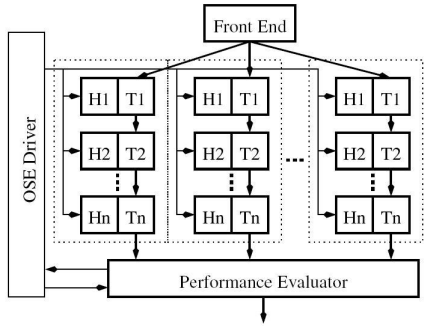


Figure 2.2: Optimization-Space Exploration approach [23]

However, the compilation time is greatly reduced by three different techniques:

1. First, the search space that is explored at compile-time is pruned at compiler construction-time and dynamically at compile-time. At construction-time, the number of configurations of optimization-parameter value pairs are limited to those that are more likely to contribute to higher performance, and these remaining configurations are arranged in a tree which will be used for the optimization search at compile-time.

2. The second technique concerns the performance evaluation function. Performance evaluation is a key factor in iterative compilation, taking an important amount of time when the execution of the produced code is needed. In [23], Triantafyllis et al. uses a static performance evaluation function that uses a machine model. Again, each target architecture will require a specific execution model.

3. The last technique consists in only considering the hot segments in the

optimization. The stated reason is that most of the execution time is spent in small portions of the code. These hot segments can be found using profiling.

The main advantage of this approach is that it can produce relatively good result with an acceptable compile time. However, the search space and the portion of optimized code are limited, which makes it unlikely to provide an optimal final code every time.

Characterization of the optimization search space

In order to make iterative compilation without having to make too many simplifications, a study of the optimization search space is necessary. Unfortunately, as written before, exhaustively enumerating all the possible optimization sequences is not feasible. However, Kulkarni et al. have been very enterprising on this topic [9, 8, 11, 12].

In [9], they developed an interactive compilation system called VISTA. This framework can be seen in Figure 2.3.

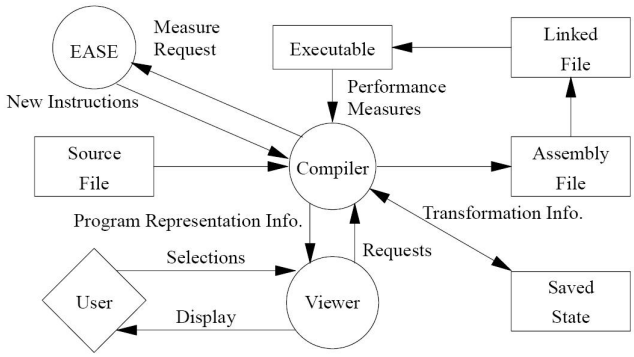


Figure 2.3: The VISTA framework. The EASE environment [6] can translate a source program to machine instructions for a proposed architecture, imitate the execution of these instructions and collect measurements.

This framework is associated to an optimization phase programming language that provides the user the opportunity to define interactively the optimization sequences he wants to apply on a specific program. Once these sequences are given, they are all evaluated, the VISTA compiler chooses the best sequence according to certain fitness criteria, and it finally displays the feedback information.

Kulkarni et al. used their VISTA framework in [8, 11] to explore the optimization search space in two different ways, while investigating the phase ordering problem without varying the optimization parameters. In [8], they used a genetic algorithm and several techniques to aggressively prune the search space in order to make the search for efficient optimization sequences faster. The definition of a genetic algorithm is considered further in this section. The pruning techniques used are:

1. Finding redundant attempted sequences: whenever a new optimization sequence is generated by mutation using the genetic algorithm, it can happen that an already attempted sequence appears. Kulkarni et al. keep track of all attempted sequence in order not to evaluate it again.

2. Finding dormant phases: some phases can have no effect when applied after other phases. By only considering active (i.e non-dormant) phases, they can find the already attempted active phases and not evaluate them again.

3. Finding identical and equivalent code: using a CRC checksum, they can detect if the resulting code has already been generated. They also detect the cases where the generated code is not identical but has the same characteristics, for example using different register names (equivalent code).

These techniques helped them to reduce the average number of generations required to find the best sequence by over two thirds.

In [11], they used these techniques to make an exhaustive enumeration of the optimization phase order space in a reasonable amount of time. The main pruning process can be seen in Figure 2.4.

Thanks to this enumeration, they were able to gather some very interesting experimental insights concerning the interactions between the optimization they used. They finally produced a probabilistic compilation algorithm using the dependency probabilities from these experimental results. This exhaustive enumeration of the optimization phase order search space finally helped them to find instances that achieve optimal dynamic execution performance in [12] for most of the program functions they considered.

Using heuristics in adaptive compilers

As seen before, the exhaustive exploration or the optimization phase order space is a good way to produce optimal optimization sequences, but, although Kulkarni et al. made it possible in a reasonable amount of time for a large majority of

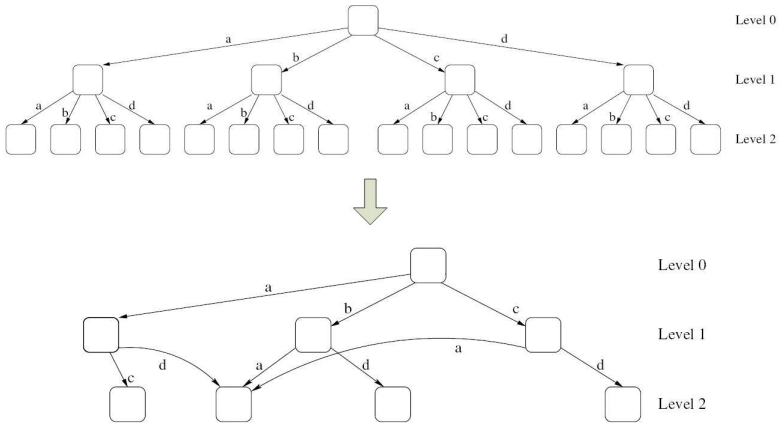


Figure 2.4: Pruning the search space [11]. The first figure shows a naive space enumeration, while the second one is reduced using pruning techniques (deleting dormant phases and joining identical instances detection).

the functions, it still takes long for most large functions, making it unsuitable for routine use in iterative compilers.

Current compilers employ most often faster heuristic algorithms in order to scan only a smaller part of the search space. The common heuristic search techniques are:

- **The local search techniques.** One example is the *hill climbing* algorithm. This algorithm starts with a randomly chosen phase sequence, and measure its performance. Then the performance of all its neighbors is evaluated. The neighbors can be defined to be all sequences that differ from the base sequence in a single position. If one of the neighbors has an equal or better performance than the base sequence, this neighbor becomes the new base sequence. This technique helps finding local optimum in the phase order space. This is repeated for several sequence lengths.
- **The greedy algorithms.** *Greedy algorithms* look for the locally optimum choice at every step of the sequence construction, in the hope of finally reaching the global optimum. The base sequence can be the empty sequence for the phase ordering problem.
- **Algorithms that focus on leaf instances.** *Genetic algorithms* are one of these. They deals with leaf instances, which are sequences that cannot be modified to change the resulting code anymore. They are based on

Darwin's theory of evolution; *genes* are the optimization phases, and the *chromosomes* are the optimization sequences. For example, in [10], after being initialized, the chromosomes are ranked by performance. The best ones are kept, while the worst ones and some randomly chosen from the poorly performing half suffer a *mutation*, where the sequence is mixed with one from a good performing chromosome.

Using these techniques permits to decrease the compilation time, but it is not possible to be ensured that the generated program will be optimal. Almagor et al. [2] performed an experimental study to evaluate some of these techniques for their compiler. Their compiler is said to be *adaptive* because it uses a steering algorithm that automatically adjusts the compilation order once the previous phase sequence has been evaluated. In order to complete their experiments, they had to restrict the space to 10-of-5 subspaces (sequences of length 10 from 5 optimizations) and small programs. Their results show that biased search can do almost as well as the more expensive genetic algorithms.

On the same topic, Kulkarni et al. [10] used their previously enumerated space to perform a full study of various heuristic search algorithms. Again, they conclude that simpler techniques like *hill climbing* (with multiple iterations) can produce better results than more complex solutions like *genetic algorithms*.

2.1.3 To a better understanding of the optimizations

In order to improve the previously described compilation techniques, understanding the different optimizations and their interactions remains one of the most useful methods though it may also be one of the most challenging. In [13], Lee et al. provides an experimental methodology to measure the benefits and the cost of optimizations, alone and in combination with the other optimizations. However, their experiments still revealed that some of the optimization phases considered had unpredictable behavior.

Aside from all the experimental work attempted to analyze the behavior of the optimizations and find an efficient solution for the phase ordering problem, some researchers have attempted to have a more formal approach in order to improve the global understanding of the optimizations and address the problem more systematically. Whitfield and Sofia [24] created a framework to specify formally some classical compiler optimizations in order to have a more theoretical and systematic approach of the problem. This framework allows to describe the optimizations and get theoretical feedback on the different inter-

actions between these transformations. In [25], they extended this framework to define a specification language, called Gospel, that can be used to describe other optimizations, as well as a tool that, given the Gospel specification of a transformation, can generate the transformation code. The main issue is that in cases of cyclic interactions between two optimizations, i.e. when two optimizations enable each other, it is not possible to know the best ordering without having specific informations concerning the compiler.

2.1.4 Conclusions

This section related the previous work in attempting to find solutions to the phase ordering problem. Currently, iterative compilers using heuristic search algorithms are the most used, because they are not evaluating all the search space, thus taking less time. These heuristics have been experimentally shown to provide efficient results, though it is of course not possible to be sure that the generating code will be optimal. Hence, despite the longer compilation time, complete iterative compilation is most often used for high performance applications, as well as final applications in embedded systems, as they need to be as optimized as possible.

Research has also produced other compilers capable to compile in a reduced compile time, thanks to the use of static performance estimators that does not require the real-time execution of the resulting code. However, the performance of the generated code may be lower than the one from iterative compilation, due to the number of assumptions and the approximation made in the performance evaluation function.

2.2 Data Flow analysis and algorithms

The analyses considered in this thesis are all following the Data-Flow Analysis approach. This section defines in what consists this approach, then describes how some analyses can be grouped into frameworks, and finally the last part considers the different algorithms available to solve the equations generated in these analyses.

2.2.1 The Data-Flow Analysis principle

In Data Flow Analysis, the program, in its intermediate representation, is seen as a *graph*. The *nodes* of this graph are the different elementary blocks of the program, and the edges represent the control flow of the program, i.e. the possible sequence in which the program's blocks can be executed. Each block is labeled in order to be directly accessible when performing the analyses.

Figure 2.6 shows the control flow for the small factorial program of Figure 2.5.

$$[\texttt{x:=5}]^1; [\texttt{y:=1}]^2; \texttt{while } [\texttt{x>1}]^3 \texttt{ do } [\texttt{y:=x*y}]^4; [\texttt{x:=x-1}]^5 \texttt{ od}$$

Figure 2.5: The factorial program.

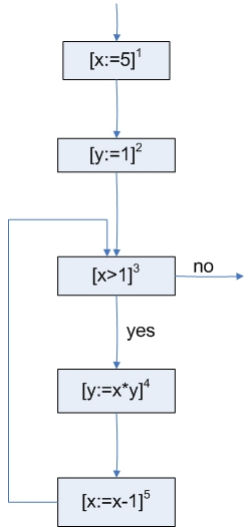


Figure 2.6: Flow graph for the factorial program.

Data-Flow Analysis provides information on specific data of the program (value of variables, reaching definitions,...) at the entry and the exit of each block. In order to compute these information, each instance of data-flow analyses defines a *transfer function*, which is a relationship between the data before a block (at the entry) and the data after the block (at the exit). Then, the definition of this transfer function on the different blocks of the program defines the *data-flow equations* for a specific analysis.

2.2.2 Monotone Frameworks

The different analyses from Data-Flow Analysis define different transfer functions. They traverse the flow graph in different ways and concern several type of data. However, there exist some similarities between them that make possible to define an underlying framework.

The Monotone Framework defines the analyses where the transfer function is a monotone function: $f_l : L \to L$. The data-flow equations for the analyses of this framework take the form

$$\begin{aligned} Analysis_{entry}(l) &= \begin{cases} \iota & \text{if } l \in E \\ \bigsqcup\{Analysis_{exit}(l') | (l', l) \in F\} & \text{otherwise} \end{cases} \\ Analysis_{exit}(l) &= f_l(Analysis_{entry}(l)) \end{aligned}$$

where

- $Analysis_{entry}(l)$ and $Analysis_{exit}(l)$ are the data-flow information at the entry and exit of the block labelled l.
- $\bigsqcup$ is $\bigcap$ or $\bigcup$ (and $\sqcup$ is $\cap$ or $\cup$).
- F defines pairs of label corresponding to the edges of the program. These edges can be either *forward* or *backward* edges. For example, for Figure 2.6, F can be either $\{(1,2),(2,3),(3,4),(4,5),(5,3)\}$ for a forward flow and $\{(2,1),(3,2),(4,3),(5,4),(3,5)\}$ for a backward flow.
- E is the entry set from which to start the equations. For Figure 2.6, E would be $\{1\}$ for a forward analysis and $\{3\}$ for a backward one.
- ι specifies the initial or final analysis information.

L is defined to be the *property space* of an analysis. This property space is used to represent the data flow information. In fact, most of the time this property space is a *complete lattice*. As defined in [16], a *complete lattice* is a partially ordered set, $(L, \sqsubseteq)$, such that each subset, Y, has a least upper bound, $\bigsqcup Y$. A subset Y has $l \in L$ as an *upper bound* if $\forall l' \in Y : l' \sqsubseteq l$, and a *least upper bound* l of Y is an upper bound of Y that satisfies $l \sqsubseteq l_0$ whenever l_0 is another upper bound of Y. Furthermore, $\bot = \bigsqcup \emptyset = \bigsqcap L$ is the *least element* (or *bottom*) and $\top = \bigsqcap \emptyset = \bigsqcup L$ is the *greatest element* (or *top*). A lattice L is often represented as a set $(L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$.

There exist stronger concepts than the Monotone Framework, including:

- A *Distributive Framework* is a Monotone Framework where additionally all functions f of F are required to be distributive:

$$f(\lambda_1 \sqcup \lambda_2) = f(\lambda_1) \sqcup f(\lambda_2)$$

- A *Bit Vector Framework* is a Distributive Framework where additionally L is a powerset of a finite set and all functions f of F have the form:

$$f(\lambda) = (\lambda \backslash kill) \cup gen$$

  where $kill$ and gen are two sets.

2.2.3 Algorithms

Here is an example of data-flow equations for Reaching Definitions Analysis (defined in Section 3.3.1). Considering the factorial program again (Figure 2.5), the $kill_{RD}$ and gen_{RD} sets are for this program:

l	$kill_{RD}(l)$	$gen_{RD}(l)$
1	$\{(x,?),(x,1),(x,5)\}$	$\{(x,1)\}$
2	$\{(y,?),(y,2),(y,4)\}$	$\{(y,2)\}$
3	$\emptyset$	$\emptyset$
4	$\{(y,?),(y,2),(y,4)\}$	$\{(y,4)\}$
5	$\{(x,?),(x,1),(x,5)\}$	$\{(x,5)\}$

The equations for this analysis are shown in Figure 2.7. Several algorithms are available to solve these equations. In the remaining of this section, two of these algorithms are described: the first one, called the *MFP solution* (for *Maximal Fixed Point*) uses the framework to obtain an analysis result, while the second one, an *abstract worklist algorithm*, is a general algorithm for solving equation and inequation systems. Other algorithms are available, for example the *MOP solution* (for *Meet Over all Paths*) defined in [16] or the *Region-Based Analysis* algorithm defined in [1].

2.2.3.1 The MFP solution

The MFP algorithm calculates the information reaching a node by combining the information from all its predecessors. Then the transfer function of the node

$$
\begin{aligned}
\mathrm{RD}_{entry}(1) &= \{(x,?),(y,?)\} \\
\mathrm{RD}_{entry}(2) &= \mathrm{RD}_{exit}(1) \\
\mathrm{RD}_{entry}(3) &= \mathrm{RD}_{exit}(2) \cup \mathrm{RD}_{exit}(5) \\
\mathrm{RD}_{entry}(4) &= \mathrm{RD}_{exit}(3) \\
\mathrm{RD}_{entry}(5) &= \mathrm{RD}_{exit}(4) \\
\mathrm{RD}_{exit}(1) &= (\mathrm{RD}_{entry}(1) \setminus \{(x,?),(x,1),(x,5)\}) \cup \{(x,1)\} \\
\mathrm{RD}_{exit}(2) &= (\mathrm{RD}_{entry}(2) \setminus \{(y,?),(y,2),(y,4)\}) \cup \{(y,2)\} \\
\mathrm{RD}_{exit}(3) &= \mathrm{RD}_{entry}(3) \\
\mathrm{RD}_{exit}(4) &= (\mathrm{RD}_{entry}(4) \setminus \{(y,?),(y,2),(y,4)\}) \cup \{(y,4)\} \\
\mathrm{RD}_{exit}(5) &= (\mathrm{RD}_{entry}(5) \setminus \{(x,?),(x,1),(x,5)\}) \cup \{(x,5)\}
\end{aligned}
$$

Figure 2.7: Reaching definition equations for the factorial program.

is applied to calculate the exit information. The MFP algorithm works iteratively: it starts from the initial block of the program and visits all blocks once or several times. Each time a block is visited, the corresponding entry and exit points are recomputed using the corresponding equations, until a fixed point is reached, i.e. until the entry and exit informations cannot change any further.

In order to know which blocks should be visited, this algorithm uses a worklist, which is initialized with the pairs of labels corresponding to all the edges of the flow graph. In every step of the algorithm, a pair is selected from the worklist. The presence of a pair in the worklist means that the analysis information has changed for the exit (or entry for *backward* analysis) of the block labeled by the first component of this pair, so the analysis informations must be recomputed for the second component of that pair. Then the pairs of the outgoing edges of the recomputed block are added to the worklist, as they point to block where information may have to be recomputed again. The algorithm continues by selecting nodes from the worklist and ends whenever there are no nodes left in it.

The pseudo-code for the MFP algorithm can be found in [16].

2.2.3.2 The Abstract Worklist Algorithm

The Abstract Worklist Algorithm is abstracting away from the details of a particular analysis. Instead, this algorithm takes as input a set of constraints that can be derived from the equation system of the analysis. The entry and exit in-

formation at each block used in Data-Flow analysis are represented as *flow variables* for which the algorithm has to solve the system of constraints. For example, the set of Reaching Definitions equations for the factorial program in Figure 2.7 can be represented as in Figure 2.8. In this figure, $X_{l_1 l_2} = \{(x, l_1), (x, l_2)\}$; $\times_1$,..., $\times_5$ correspond to $\mathrm{RD}_{entry}(1)$,..., $\mathrm{RD}_{entry}(5)$; and $\times_6$,..., $\times_{10}$ correspond to $\mathrm{RD}_{exit}(1)$,..., $\mathrm{RD}_{exit}(5)$.

$$
\begin{array}{lcllcl}
\times_1 & = & X_? \cup Y_? & \times_6 & = & (\times_1 \backslash X_{15?}) \cup X_1 \\
\times_2 & = & \times_6 & \times_7 & = & (\times_2 \backslash Y_{24?}) \cup Y_2 \\
\times_3 & = & \times_7 \cup \times_{10} & \times_8 & = & \times_3 \\
\times_4 & = & \times_8 & \times_9 & = & (\times_4 \backslash Y_{24?}) \cup Y_4 \\
\times_5 & = & \times_9 & \times_{10} & = & (\times_5 \backslash X_{15?}) \cup X_5
\end{array}
$$

Figure 2.8: Reaching definition equations for the factorial program.

As the main interest remains in RD_{entry}, the equations can be simplified as in Figure 2.9, without losing any information.

$$
\begin{array}{lcl}
\times_1 & = & X_? \cup Y_? \\
\times_2 & = & (\times_1 \backslash X_{15?}) \cup X_1 \\
\times_3 & = & (\times_2 \backslash Y_{24?}) \cup Y_2 \cup (\times_5 \backslash X_{15?}) \cup X_5 \cup Y_2 \\
\times_4 & = & \times_3 \\
\times_5 & = & (\times_4 \backslash Y_{24?}) \cup Y_4
\end{array}
$$

Figure 2.9: Simplified reaching definition equations.

The Abstract Worklist Algorithm works in the same way as the MFP algorithm: all the constraints are initially placed into a worklist; at each iteration the first constraint is extracted from the worklist and evaluated, and if there is a change on the flow variable's value, the other constraints influenced by that flow variable are put in the worklist again. These iterations last until no changes occur anymore, i.e. the worklist becomes empty.

The variations on this algorithm concern the way the constraints are organized in the worklist. Indeed, the worklist could be simply represented as a set where constraints are taken at random, or some more sophisticated representation can be used, like using a *reverse postorder* organization or grouping the constraints in *strong components*, as described in [16]. These more advanced technique are

more difficult to implement, but they allow to decrease the number of iterations of the algorithm and thus to get a better performance.

These two algorithms are widely used to solve data flow analyses' equations, and have been implemented in the WHILE compiler used in this thesis. They will thus be considered during the analysis of the phase-ordering problem and the design of the metrics, and compared to a new algorithm, called the *propagation algorithm*, defined in Chapter 5.

Chapter 3

Setting the Scene

The goal of this chapter is to make the reader familiar with the environment in which the phase-ordering problem has been addressed, and to enable him to understand the various references to the different elements of the compiler. This chapter first describes the simple imperative language which will be compiled by the implemented tool. Then it gives a brief overview of the different elements involved in the WHILE compiler. Finally, the different analyses and transformations implemented in the compiler are described.

3.1 Description of the WHILE language

The language used is based on the WHILE language described in [16]. This is a good example of a simple imperative programming language but complete enough to act as a basis to analyze more complex languages like C. Programs in WHILE are, moreover, easily convertible into C programs.

The description of the language (which will be called the WHILE language in this thesis, for the sake of simplicity) is given in Figure 3.1 where x is a variable, n is a natural number, op_b is a binary operator from the set $\{+, -, *, /, \&, |, =, <, <=, >, >=, !=\}$ and op_m is a monadic operator from the set $\{\neg, -\}$.

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid \text{true} \mid \text{false} \mid e_1\ op_b\ e_2 \mid op_m\ e \mid m(e) \\
S & ::= & [x := e]^l \mid [\text{skip}]^l \mid S_1; S_2 \mid \text{begin } D\ S \text{ end} \mid \\
 & & \text{if } [e]^l \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{if } [e]^l \text{ then } S \text{ fi} \mid \\
 & & \text{while } [e]^l \text{ do } S \text{ od} \mid [\text{read } x]^l \mid \text{write } [e]^l \\
D & ::= & \text{var } x; D \mid \epsilon \\
M & ::= & \text{func var } m(\text{var } x) \text{ begin } D\ S; \text{return } e \text{ end}; M \mid \epsilon \\
P & ::= & D; M; S
\end{array}
$$

Figure 3.1: The WHILE language

The value l represents the label of a specific block, and is used to represent the program by a flow graph.

3.2 Elements of the WHILE language compiler

The WHILE compiler is composed by a lexical analyzer and a parser that represent the frontend of the compiler; an optimization module, and a backend composed by an interpreter and a deparser.
The lexical analyzer and the parser are both generated by tools and written in Java. Figure 3.2 represents the overall structure of the WHILE compiler.

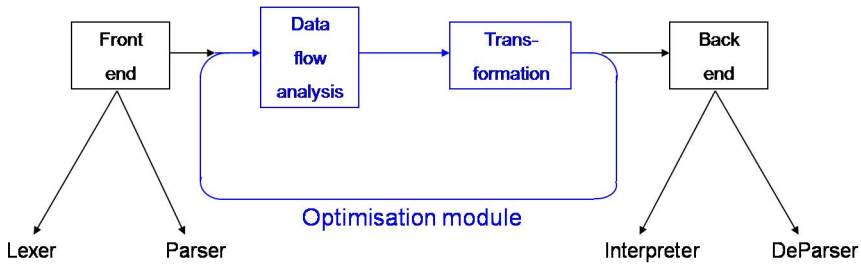


Figure 3.2: Structure of the WHILE compiler

3.2.1 The frontend and the intermediate representation

The lexical analyzer is used to tokenize any WHILE program described with the given syntax into an appropriate sequence of symbol. It takes as input a description of the tokens to be generated, which are in fact the terminals of the WHILE language. Once the lexer has tokenized the program, it needs to be parsed into a parse tree. This parse tree will be the internal representation of the parsed program, on which the analyses and transformations will be applied.

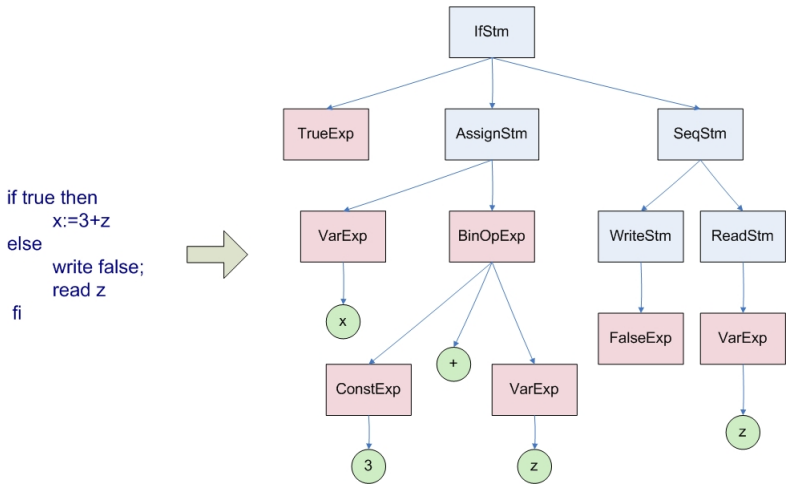


Figure 3.3: Example of a parse tree

Figure 3.3 shows an example of parsing a statement into a tree. The blue nodes of the parse tree represent instances of statements, while the pink nodes are instances of expressions. Finally the round green nodes are terminals of the program.

Moreover, all the expressions and the statements contain a reference to their parent statement (field *up*) in order to be able to redefine these expressions when performing transformations in the program: this reference corresponds to a reverse edge in the parse tree, and allows to go up in the parse tree when needed. Finally, a reference to the *scope* the component belongs to has also been added to all the components of the abstract syntax, as well as the labels for some specific statements (assign, read and skip).

3.2.2 Interpreter

Instead of providing a backend that transforms the intermediate representation into machine code, an Interpreter module has been designed. It can interpret the program and write any results according to the semantics of the WHILE language defined in [17]. This module goes through the parse tree to update some variable environments and stores where the values of the different variables in the program are stored. The Interpreter can also define some type of errors, as a type mismatch or the case where a variable has been used before being initialized in order to stop the execution of the program.

An interesting feature of the Interpreter is also to indicate the number of instructions executed when the program is interpreted. Each time an instruction is executed, a counter is incremented. However, all the instructions do not have the same weight, in order to have a better estimation of the performance of the program. For example, reading a variable x will have a higher weight than reading a constant n. This weighted number of instructions executed can be very interesting when comparing the different order of optimizations applied during the different benchmarks. The different weights can of course be adjusted easily.

3.2.3 DeParser

A DeParser module can output the whole program after the optimization process. It translates the intermediate representation of the program into WHILE language.

The main interest in this module is to get an understandable output of the program after having applied a series of optimizations, in order to ensure the correctness of the optimized program as well as its level of optimization.

3.3 Data Flow Analysis

In the WHILE compiler, eight different analyses have been implemented:

1. the Reaching Definitions Analysis
2. the Available Expressions Analysis
3. the Copy Analysis

4. the Very Busy Expressions Analysis
5. the Live Variable Analysis
6. the Strongly Live Variables Analysis
7. the Constant Propagation Analysis
8. the Detection of Signs Analysis

The first five analyses are instances of the *Bit Vector Framework* (described in Section 2.2.2), thus their transfer function is following the same pattern:

$$f(\lambda) = (\lambda \backslash kill(B^l)) \cup gen(B^l) \text{ where } B^l \in blocks(S_*)$$

The Strongly Live Variables is just an instance of the Distributive Framework, while the last two analysis are only instances of the Monotone Framework.

In the remainder of this section all the parameters of the lattices specifying these eight different analyses will be described.

3.3.1 Reaching Definitions Analysis

The first analysis is the Reaching Definitions analysis. The aim of this analysis is to determine for each program point, which assignments *may* have been made and not overwritten, when program execution reaches this point along some path. In summary, the analysis gives, at each program block entry or exit, for each variable occurring in the program, the (smallest possible) set of labels where the variable *may* have obtained its value. The special token ? is used to indicate that the variable may have obtained its value outside the program.

The Reaching Definitions analysis is a forward, over-approximation analysis. As an instance of the Bit Vector Framework, the parameters of the lattice are:

- $\mathcal{P}(\mathbf{Var}_* \times (\mathbf{Lab}_*^?))$, the lattice of variables and labels, indicating where a variable is last defined
- a partial ordering $\subseteq$ and least upper bound operation $\bigcup$ that since it is a *may*-analysis
- $\emptyset$, the least element of the lattice $\bot$
- $\{(x, ?) \mid x \in FV(S_*)\}$, the extremal value ι of the analysis

- $\{init(S_*)\}$, the extremal labels E
- $flow(S_*)$, the definition of the flow F through the program
- the transition function f of the Bit Vector Framework

Finally, we need to define the *kill* and *gen* functions used in the transition function f_l:

$$\begin{aligned} kill_{RD}([x := a]^l) &= \{(x, ?)\} \cup \{(x, l') \mid B^{l'} \text{ is an assignment to } x \text{ in } S_*\} \\ kill_{RD}([\text{read } x]^l) &= \{(x, ?)\} \cup \{(x, l') \mid B^{l'} \text{ is an assignment to } x \text{ in } S_*\} \\ gen_{RD}([x := a]^l) &= \{(x, l)\} \\ gen_{RD}([\text{read } x]^l) &= \{(x, l)\} \\ kill_{RD}(B^l) &= gen_{RD}(B^l) = \emptyset \quad \text{otherwise} \end{aligned}$$

Consider the following program:

$$[\texttt{x:=a+b}]^1; [\texttt{y:=a*b}]^2; \texttt{while } [\texttt{y>a+b}]^3 \texttt{ do } [\texttt{a:=a+1}]^4; [\texttt{x:=a+b}]^5 \texttt{ od}$$

Figure 3.4: Example program 1

Thanks to the framework defined above, the Reaching Definitions solutions can be computed for the program in Figure 3.4:

l	$\text{RD}_{entry}(l)$	$\text{RD}_{exit}(l)$
1	$\{(\texttt{x}, ?), (\texttt{y}, ?), (\texttt{a}, ?)\}$	$\{(\texttt{x}, 1), (\texttt{y}, ?), (\texttt{a}, ?)\}$
2	$\{(\texttt{x}, 1), (\texttt{y}, ?), (\texttt{a}, ?)\}$	$\{(\texttt{x}, 1), (\texttt{y}, 2), (\texttt{a}, ?)\}$
3	$\{(\texttt{x}, 1), (\texttt{x}, 5), (\texttt{y}, 2), (\texttt{a}, ?), (\texttt{a}, 4)\}$	$\{(\texttt{x}, 1), (\texttt{x}, 5), (\texttt{y}, 2), (\texttt{a}, ?), (\texttt{a}, 4)\}$
4	$\{(\texttt{x}, 1), (\texttt{x}, 5), (\texttt{y}, 2), (\texttt{a}, ?), (\texttt{a}, 4)\}$	$\{(\texttt{x}, 1), (\texttt{x}, 5), (\texttt{y}, 2), (\texttt{a}, 4)\}$
5	$\{(\texttt{x}, 1), (\texttt{x}, 5), (\texttt{y}, 2), (\texttt{a}, 4)\}$	$\{(\texttt{x}, 5), (\texttt{y}, 2), (\texttt{a}, 4)\}$

3.3.2 Use-Definition and Definition-Use chains

Before looking at other analysis, it is interesting to define two functions, called the *Use-Definition chain* (often abbreviated ud-chain) and the *Definition-Use chain* (abbreviated du-chain). The Use-Definition chain links each use of a variable to the different assignments that could have define it, while the Definition-Use chain links an assignment defining a variable to all the potential uses of this

variable.

It is possible to obtain the ud-chain from the results of the Reaching Definitions Analysis described in the previous section ($\mathrm{RD}_\circ(l)$ is an abbreviation for $\mathrm{RD}_{entry}(l)$):

$$UD : \mathbf{Var}_* \times \mathbf{Lab}_* \to \mathcal{P}(\mathbf{Lab}_*)$$

$$UD(x,l) = \begin{cases} \{l' | (x,l') \in \mathrm{RD}_\circ(l)\} & \text{if } x \in \mathrm{used}(B^l) \\ \emptyset & \text{otherwise} \end{cases}$$

with

$$\mathrm{used}([x := a]^l) = FV(a),\ \mathrm{used}([b]^l) = FV(b)$$
$$\mathrm{used}([\mathrm{read}\ x]^l) = \mathrm{used}([skip]^l) = \emptyset$$

and FV: $\mathbf{AExp} \to \mathbf{Var}_*$ is a function returning the set of variables occurring in an arithmetic expression.

The du-chain can be computed directly from the ud-chain:

$$DU : \mathbf{Var}_* \times \mathbf{Lab}_* \to \mathcal{P}(\mathbf{Lab}_*)$$
$$DU(x,l) = \{l' | l \in UD(x,l')\}$$

3.3.3 Available Expressions Analysis

The aim of the Available Expressions Analysis is to determine for each program point, which expressions *must* have already been computed, and not later modified, on all paths to the program point.

The Available Expressions Analysis is a forward, under-approximation analysis which is an instance of the Bit Vector Framework. The parameters for the lattice are:

- $\mathcal{P}(\mathbf{AExp}_*)$, the lattice of all non-trivial arithmetic expressions occurring in the program
- a partial ordering $\supseteq$ and least upper bound operation $\bigcap$ that since this is a *must*-analysis

- $\mathbf{AExp}_*$, the least element of the lattice $\perp$
- $\emptyset$, the extremal value ι of the analysis
- $\{init(S_*)\}$, the extremal labels E
- $flow(S_*)$, the definition of the flow F through the program
- the transition function f of the Bit Vector Framework

We also need the definitions of the *kill* and *gen* functions used in the transition function f_l:

$$\begin{aligned}
kill_{AE}([x := a]^l) &= \{a' \in \mathbf{AExp}_* \mid x \in FV(a')\} \\
kill_{AE}([\text{read } x]^l) &= \{a' \in \mathbf{AExp}_* \mid x \in FV(a')\} \\
kill_{AE}([b]^l) &= \emptyset \\
kill_{AE}([skip]^l) &= \emptyset \\
gen_{AE}([x := a]^l) &= \{a' \in \mathbf{AExp}(a) \mid x \notin FV(a')\} \\
gen_{AE}([b]^l) &= \mathbf{AExp}(b) \\
gen_{AE}([\text{read } x]^l) &= \emptyset \\
gen_{AE}([skip]^l) &= \emptyset
\end{aligned}$$

For example, it is possible to compute the Available Expressions Analysis solutions for the program in Figure 3.4:

l	$\text{AE}_{entry}(l)$	$\text{AE}_{exit}(l)$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b,a*b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}

3.3.4 Very Busy Expressions Analysis

An expression is *very busy* at the exit from a label if, no matter what path is taken from the label, the expression is always used before any of the variables occurring in it are redefined.
The aim of the Very Busy Expressions Analysis is to determine, for each program point, which expressions must be *very busy* at the exit from the point.

The Very Busy Expressions Analysis is a backward, under-approximation analysis, and an instance of the Bit Vector Framework. The parameters of the lattice are:

- $\mathcal{P}(\mathbf{AExp}_*)$, the lattice of all non-trivial arithmetic expressions occurring in the program
- a partial ordering $\supseteq$ and least upper bound operation $\bigcap$ that since this is a *must*-analysis
- $\mathbf{AExp}_*$, the least element of the lattice $\bot$
- $\emptyset$, the extremal value ι of the analysis
- $final(S_*)$, the extremal labels E
- $flow^R(S_*)$, the definition of the flow F through the program
- the transition function f of the Bit Vector Framework

The definitions of the *kill* and *gen* functions are:

$$\begin{aligned}
kill_{VB}([x := a]^l) &= \{a' \in \mathbf{AExp}_* \mid x \in FV(a')\} \\
kill_{VB}([\text{read } x]^l) &= \{a' \in \mathbf{AExp}_* \mid x \in FV(a')\} \\
kill_{VB}([b]^l) &= \emptyset \\
kill_{VB}([skip]^l) &= \emptyset \\
gen_{VB}([x := a]^l) &= \mathbf{AExp}(a) \\
gen_{VB}([b]^l) &= \mathbf{AExp}(b) \\
gen_{VB}([\text{read } x]^l) &= \emptyset \\
gen_{VB}([skip]^l) &= \emptyset
\end{aligned}$$

For example, it is possible to compute the Available Expressions Analysis solutions for the program in Figure 3.4:

l	$\text{VB}_{entry}(l)$	$\text{VB}_{exit}(l)$
1	{a+b,a*b}	{a+b,a*b}
2	{a+b,a*b}	{a+b}
3	{a+b}	$\emptyset$
4	{a+1}	{a+b}
5	{a+b}	{a+b}

3.3.5 Copy Analysis

The aim of the Copy Analysis is to determine for each program point, which copy statements $[x := y]^l$ still are relevant i.e., neither x nor y have been redefined, when control reaches that point.

The Copy Analysis is a forward, under-approximation analysis and an instance of the Bit Vector Framework. The parameters for the lattice are:

- $\mathcal{P}(\mathbf{Var}_* \times \mathbf{Var}_*)$, the lattice of all pairs of variables occurring in the program
- a partial ordering $\supseteq$ and least upper bound operation $\bigcap$ that since this is a *must*-analysis
- $\mathbf{Var}_* \times \mathbf{Var}_*$, the least element of the lattice $\perp$
- $\emptyset$, the extremal value ι of the analysis
- $\{init(S_*)\}$, the extremal labels E
- $flow(S_*)$, the definition of the flow F through the program
- the transition function f of the Bit Vector Framework

The definitions of the *kill* and *gen* functions used in the transition function f_l are:

$$\begin{aligned} kill_{CA}([x := a]^l) &= \{(x,y)|y \in FV(S^*)\} \cup \{(y,x)|y \in FV(S^*)\} \\ kill_{CA}([\text{read } x]^l) &= \{(x,y)|y \in FV(S^*)\} \cup \{(y,x)|y \in FV(S^*)\} \\ gen_{CA}([x := a]^l) &= \{(x,y)|a = y \wedge y \in FV(S^*)\} \\ kill_{CA}(B^l) &= gen_{CA}(B^l) = \emptyset \quad \text{otherwise} \end{aligned}$$

Consider another example program:

$$[\texttt{x:=1}]^1; [\texttt{y:=x}]^2; [\texttt{z:=y*(-y)}]^3; \texttt{write } [\texttt{y}]^4; [\texttt{y:=10}]^5$$

Figure 3.5: Example program 2

Thanks to the framework defined above, the Copy Analysis solutions can be computed for the program in Figure 3.5:

l	$\text{CA}_{entry}(l)$	$\text{CA}_{exit}(l)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{(\mathtt{x},\mathtt{y})\}$
3	$\{(\mathtt{x},\mathtt{y})\}$	$\{(\mathtt{x},\mathtt{y})\}$
4	$\{(\mathtt{x},\mathtt{y})\}$	$\{(\mathtt{x},\mathtt{y})\}$
5	$\{(\mathtt{x},\mathtt{y})\}$	$\emptyset$

3.3.6 Live Variables Analysis

A variable is *live* at the exit from a label if there is a path from the label to a use of the variable that does not re-define the variable. The aim of the Live Variables Analysis is to determine for each program point, which variables may be *live* at the exit from the point. In practice, we are interested in variables that are not live at the exit from a program point.

The Live Variables Analysis is a backward, over-approximation analysis and an instance of the Bit Vector Framework. The parameters for the lattice are:

- $\mathcal{P}(\mathbf{Var}_*)$, the lattice of all variables occurring in the program
- a partial ordering $\subseteq$ and least upper bound operation $\bigcup$ that since this is a *may*-analysis
- $\emptyset$, the least element of the lattice $\bot$
- $\emptyset$, the extremal value ι of the analysis
- $final(S_*)$, the extremal labels E
- $flow^R(S_*)$, the definition of the flow F through the program
- the transition function f of the Bit Vector Framework

The definitions of the *kill* and *gen* functions used in the transition function f_l are:

$$\begin{aligned} kill_{LV}([x := a]^l) &= \{x\} \\ kill_{LV}([\text{read } x]^l) &= \{x\} \\ kill_{LV}([b]^l) &= kill_{LV}([skip]^l) = \emptyset \end{aligned}$$

$$\begin{aligned} gen_{LV}([x := a]^l) &= FV(a) \\ gen_{LV}([b]^l) &= FV(b) \\ gen_{LV}([\text{read } x]^l) &= \emptyset \\ gen_{LV}([skip]^l) &= \emptyset \end{aligned}$$

For example, it is possible to compute the Live Variables Analysis solutions for the program in Figure 3.5:

l	$\text{LV}_{entry}(l)$	$\text{LV}_{exit}(l)$
1	$\emptyset$	{x}
2	{x}	{y}
3	{y}	{y}
4	{y}	$\emptyset$
5	$\emptyset$	$\emptyset$

3.3.7 Strongly Live Variables Analysis

If we consider the program:

$$[x := 1]^1; [x := x - 1]^2; [x := 2]^3$$

We can see that x is *dead* at the exits from 2 and 3. But x is live at the exit of 1 even though its only use is to calculate a new value for a variable that turns out to be *dead*.
A variable is defined as a *faint* variable if it is *dead* or if it is only used to calculate new values for *faint* variables; otherwise it is *strongly live*. In the example x is *faint* at the exit from 1.
The aim of the Strongly Live Variables Analysis is to determine which variables may be *strongly live* at the exit from a point. In practice, we are more interested in the *faint* variables which can be eliminated.

The Strongly Live Variables Analysis is *not* an instance of the Bit Vector Framework, but only of the Distributive Framework. All the parameters of the lattice are the same as the ones of the Live Variable Analysis, except the transition function f. Hence, we have to give its data flow equations:

$$SLV_{entry}(l) = \begin{cases} (SLV_{exit}(l) \backslash kill_{LV}(B^l)) \cup gen_{LV}(B^l) & \text{if } kill_{LV}(B^l) \subseteq SLV_{exit}(l) \\ SLV_{exit}(l) & \text{otherwise} \end{cases}$$

$$SLV_{exit}(l) = \bigcup \{\ SLV_{entry}(l' \mid (l', l) \in flow^R(S_*))\ \}$$

We can underline the fact that the *kill* and *gen* functions are also the same as those of the Live Variables Analysis

3.3.8 Constant Propagation Analysis

Another analysis which does not belong to the Bit Vector Framework is the Constant Propagation Analysis. The aim of this analysis is to determine, for each program point, whether or not a variable has a constant value whenever the execution reaches that point. This analysis then gives, for each program point, a mapping between the variables and the corresponding information i.e, whether or not the value of the variable is constant, and if so what is the value. We can see here the complete lattice used for the Constant Propagation Analysis, which is a forward analysis and an instance of the Monotone Framework, though not of the Distributive Framework:

- $\widehat{\mathbf{State}}_{CP} = (\mathbf{Var}_* \to \mathbf{Z}^\top)_\perp$, the lattice which maps variables to values, where $\top$ is used to indicate that a variable is not constant and $\perp$ when we have no information at all.
- $\lambda x.\top$, the extremal value ι of the analysis
- $\{init(S_*)\}$, the extremal labels E
- $flow(S_*)$, the definition of the flow F through the program
- a special partial ordering $\sqsubseteq$ defined by:

$$\begin{aligned} \forall \widehat{\sigma} \in (\mathbf{Var}_* \to \mathbf{Z}^\top)_\perp &: \perp \sqsubseteq \widehat{\sigma} \\ \forall \widehat{\sigma}_1, \widehat{\sigma}_2 \in \mathbf{Var}_* \to \mathbf{Z}^\top &: \widehat{\sigma}_1 \sqsubseteq \widehat{\sigma}_2 \text{ iff } \forall x : \widehat{\sigma}_1(x) \sqsubseteq \widehat{\sigma}_2(x) \end{aligned}$$

using the partial ordering define on $Z^\top = Z \cup \{\top\}$:

$$\begin{aligned} \forall z \in Z^\top &: z \sqsubseteq \top \\ \forall z_1, z_2 \in Z &: (z_1 \sqsubseteq z_2) \Leftrightarrow (z_1 = z_2) \end{aligned}$$

- a least upper bound operation $\sqcup$ defined by:

$$\begin{array}{rcl} \forall \widehat{\sigma} \in (\mathbf{Var}_* \to \mathbf{Z}^\top)_\perp & : & \widehat{\sigma} \sqcup \perp = \widehat{\sigma} = \perp \sqcup \widehat{\sigma} \\ \forall \widehat{\sigma}_1, \widehat{\sigma}_2 \in \mathbf{Var}_* \to \mathbf{Z}^\top & : & \forall x : (\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2)(x) = \widehat{\sigma}_1(x) \sqcup \widehat{\sigma}_2(x) \end{array}$$

- a special transfer function f_l^{CP} defined by:

$$\begin{array}{rrcl} [x := a]^l : & f_l^{CP}(\widehat{\sigma}) = & \left\{ \begin{array}{cc} \perp & \text{if } \widehat{\sigma} = \perp \\ \widehat{\sigma}[x \mapsto A_{CP}(a)_{\widehat{\sigma}}] & \text{otherwise} \end{array} \right. \\ [\text{read } x]^l : & f_l^{CP}(\widehat{\sigma}) = & \left\{ \begin{array}{cc} \perp & \text{if } \widehat{\sigma} = \perp \\ \widehat{\sigma}[x \mapsto \top] & \text{otherwise} \end{array} \right. \\ [skip]^l : & f_l^{CP}(\widehat{\sigma}) = & \widehat{\sigma} \\ [b]^l : & f_l^{CP}(\widehat{\sigma}) = & \widehat{\sigma} \end{array}$$

This transfer function uses the auxiliary function A_{CP} defined below:

$$\begin{array}{rcl} A_{CP}(x)_{\widehat{\sigma}} & = & \left\{ \begin{array}{cc} \perp & \text{if } \widehat{\sigma} = \perp \\ \widehat{\sigma}(x) & \text{otherwise} \end{array} \right. \\ A_{CP}(n)_{\widehat{\sigma}} & = & \left\{ \begin{array}{cc} \perp & \text{if } \widehat{\sigma} = \perp \\ n & \text{otherwise} \end{array} \right. \\ A_{CP}(a_1\ op_a\ a_2)_{\widehat{\sigma}} & = & A_{CP}(a_1)_{\widehat{\sigma}}\ \widehat{op_a}\ A_{CP}(a_2)_{\widehat{\sigma}} \end{array}$$

where $\widehat{op_a}$ is a function that performs a specific operation on the values given as inputs and compute the result ; if one of the operand is $\perp$ or $\top$, the value returned is $\top$.

It is now possible to compute the Constant Propagation Analysis solutions for the program in Figure 3.5:

l	$\text{CP}_{entry}(l)$	$\text{CP}_{exit}(l)$
1	$\widehat{\sigma} = \{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \top, \}$	$\widehat{\sigma} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \top, \}$
2	$\widehat{\sigma} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \top, \}$	$\widehat{\sigma} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto \top, \}$
3	$\widehat{\sigma} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto \top, \}$	$\widehat{\sigma} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto -1, \}$
4	$\widehat{\sigma} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto -1, \}$	$\widehat{\sigma} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto -1, \}$
5	$\widehat{\sigma} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto -1, \}$	$\widehat{\sigma} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 10, \mathtt{z} \mapsto -1, \}$

3.3.9 Detection of Signs Analysis

The last analysis, which does also not belong to the Bit Vector Framework is the Detection of Signs Analysis. The aim of this analysis is to decide whether a variable at a given program point will have a negative value, a positive value, be zero, or combinations of these. The analysis must specify all the signs that a variables might have.
The Detection of Signs Analysis is a forward over-approximation analysis, and an instance of the Monotone Framework:

- $\widehat{\mathbf{State}}_{DS} = (\mathbf{Var}_* \to \mathbf{Sign})$, the lattice which maps variables to a set of signs $\mathbf{Sign} = P(\{\text{-}, 0, +\})$.
- $\lambda x.\{-, 0, +\}$, the extremal value ι of the analysis
- $\{init(S_*)\}$, the extremal labels E
- $flow(S_*)$, the definition of the flow F through the program
- a special partial ordering $\sqsubseteq$ defined by:

$$\forall \widehat{\sigma}_1, \widehat{\sigma}_2 \in \mathbf{Var}_* \to \mathbf{Z}^\top \quad : \quad \widehat{\sigma}_1 \sqsubseteq \widehat{\sigma}_2 \text{ iff } \forall x : \widehat{\sigma}_1(x) \subseteq \widehat{\sigma}_2(x)$$

- a least upper bound operation $\sqcup$ defined by:

$$\forall \widehat{\sigma}_1, \widehat{\sigma}_2 \in \mathbf{Var}_* \to \mathbf{Z}^\top \quad : \quad \forall x : (\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2)(x) = \widehat{\sigma}_1(x) \cup \widehat{\sigma}_2(x)$$

- a special transfer function f_l^{DS} defined by:

$$\begin{aligned}
[x := a]^l : \quad & f_l^{DS}(\widehat{\sigma}) = \quad \widehat{\sigma}[x \mapsto A_{DS}(a)_{\widehat{\sigma}}] \\
[\text{read } a]^l : \quad & f_l^{DS}(\widehat{\sigma}) = \quad \widehat{\sigma}[x \mapsto \{-, 0, +\}] \\
[skip]^l : \quad & f_l^{DS}(\widehat{\sigma}) = \quad \widehat{\sigma} \\
[b]^l : \quad & f_l^{DS}(\widehat{\sigma}) = \quad \widehat{\sigma}
\end{aligned}$$

This transfer function uses the auxiliary function A_{DS} defined below:

$$\begin{aligned}
A_{DS}(x)_{\widehat{\sigma}} &= \widehat{\sigma}(x) \\
A_{DS}(n)_{\widehat{\sigma}} &= \begin{cases} \{-\} & \text{if } n < 0 \\ \{0\} & \text{if } n = 0 \\ \{+\} & \text{if } n > 0 \end{cases} \\
A_{DS}(a_1 \ op_a \ a_2)_{\widehat{\sigma}} &= A_{DS}(a_1)_{\widehat{\sigma}} \ \widehat{op_a} \ A_{DS}(a_2)_{\widehat{\sigma}}
\end{aligned}$$

where $\widehat{op_a}$: **Sign** $\times$ **Sign** $\rightarrow$ **Sign** is a function that performs a specific operation on the signs given as inputs and returns the corresponding results ,e.g if $\widehat{*}$, if {+} and {-} are given as inputs, it will return {-}; if {0, +} and {-} are given as inputs, it will return {-, 0}; etc...

Here are the solutions for the Detection of Signs Analysis, for the program in Figure 3.5:

l	$\text{DOS}_{entry}(l)$	$\text{DOS}_{exit}(l)$
1	$\{\mathtt{x} \mapsto \mathcal{S}_{-,0,+}, \mathtt{y} \mapsto \mathcal{S}_{-,0,+}, \mathtt{z} \mapsto \mathcal{S}_{-,0,+},\}$	$\{\mathtt{x} \mapsto \mathcal{S}_{+}, \mathtt{y} \mapsto \mathcal{S}_{-,0,+}, \mathtt{z} \mapsto \mathcal{S}_{-,0,+},\}$
2	$\{\mathtt{x} \mapsto \mathcal{S}_{+}, \mathtt{y} \mapsto \mathcal{S}_{-,0,+}, \mathtt{z} \mapsto \mathcal{S}_{-,0,+},\}$	$\{\mathtt{x} \mapsto \mathcal{S}_{+}, \mathtt{y} \mapsto \mathcal{S}_{+}, \mathtt{z} \mapsto \mathcal{S}_{-,0,+},\}$
3	$\{\mathtt{x} \mapsto \mathcal{S}_{+}, \mathtt{y} \mapsto \mathcal{S}_{+}, \mathtt{z} \mapsto \mathcal{S}_{-,0,+},\}$	$\{\mathtt{x} \mapsto \mathcal{S}_{+}, \mathtt{y} \mapsto \mathcal{S}_{+}, \mathtt{z} \mapsto \mathcal{S}_{-},\}$
4	$\{\mathtt{x} \mapsto \mathcal{S}_{+}, \mathtt{y} \mapsto \mathcal{S}_{+}, \mathtt{z} \mapsto \mathcal{S}_{-},\}$	$\{\mathtt{x} \mapsto \mathcal{S}_{+}, \mathtt{y} \mapsto \mathcal{S}_{+}, \mathtt{z} \mapsto \mathcal{S}_{-},\}$
5	$\{\mathtt{x} \mapsto \mathcal{S}_{+}, \mathtt{y} \mapsto \mathcal{S}_{+}, \mathtt{z} \mapsto \mathcal{S}_{-},\}$	$\{\mathtt{x} \mapsto \mathcal{S}_{+}, \mathtt{y} \mapsto \mathcal{S}_{+}, \mathtt{z} \mapsto \mathcal{S}_{-},\}$

where $\mathcal{S}_{-,0,+} = \{-, 0, +\}$, $\mathcal{S}_{+} = \{+\}$, etc...

3.4 Transformations performed on the program

This section describes the eight transformations implemented in the WHILE compiler:

1. the Constant Folding transformation
2. the Common Subexpressions Elimination transformation
3. the Copy Propagation transformation
4. the Dead Code Elimination transformation
5. the Code Motion transformation
6. the Elimination with Signs transformation
7. the Precalculation process

3.4.1 Constant Folding

When the Reaching Definitions analysis is performed, we can use the results to optimize the program using Constant Folding. The aim is to find where the assignment of a variable x in the block l $[\texttt{x:=n}]^l$ is used in the later blocks, before any other assignment of x.
Then, in the concerned blocks, we can replace the variable x by its value n. In the case that there are several assignments of x that reach one block l', the transformation is also effective if n is assigned to x in every one of them. In this situation, we use the ud-chains.
Example
Consider this small program :

$$[\texttt{x:=4}]^1; [\texttt{x:=5}]^2; [\texttt{y:=x+5}]^3$$

becomes

$$[\texttt{x:=4}]^1; [\texttt{x:=5}]^2; [\texttt{y:=5+5}]^3$$

after the Constant Folding Transformation.
This transformation does not add or remove lines of the program, it only reduces the number of access to the variables by replacing them by their value when it can be known.

This transformation can also be applied using the Constant Propagation Analysis. As this analysis directly provides the value of a variable (if it is constant), this second implementation is more straightforward. Indeed, it is enough to visit all the variables used in each blocks of the program and, in the case the map resulting of the Constant Propagation Analysis contains a constant value for this variable, perform the change.

Not only the implementation of Constant Folding is easier using this analysis, but the results are also better. Indeed, the Constant Propagation Analysis includes a static computation and a propagation of the constant value of variables, while using the Reaching Definitions Analysis it is sometimes necessary to apply the Constant Folding transformation several times.

Example
Consider the following example :

$$[\texttt{x:=4}]^1; [\texttt{y:=x*x}]^2; \texttt{write } [\texttt{y}]^3$$

becomes, using Constant Folding with Reaching Definitions Analysis

$$[\texttt{x:=4}]^1; [\texttt{y:=4*4}]^2; \texttt{write } [\texttt{y}]^3$$

while it becomes, using Constant Folding with Constant Propagation Analysis

$$[\texttt{x:=4}]^1; [\texttt{y:=4*4}]^2; \texttt{write } [\texttt{16}]^3$$

3.4.2 Common Subexpressions Elimination

Using the Available Expressions Analysis, we can perform a new transformation: the Common Subexpressions Elimination transformation. The aim of that optimization is to find the expressions used at least twice during a path of the program and to compute them once before any use. Then, instead of calculating twice or more the same expression, it will be computed only once, hence a possible gain of time, even if, after this transformation, the program may be longer (in number of statements) with more variables.

Example
The program :

$$[\texttt{x:=a+b}]^1; [\texttt{y:=3}]^2; \texttt{if } [\texttt{y<a+b}]^3 \texttt{ then } [\texttt{y:=a+b}]^4; [\texttt{x:=b}]^5 \texttt{ fi};$$

becomes

$$[\texttt{u:=a+b}];^1 [\texttt{x:=u}]^2; [\texttt{y:=3}]^3; \texttt{if } [\texttt{y<u}]^4 \texttt{ then } [\texttt{y:=u}]^5; [\texttt{x:=b}]^6 \texttt{ fi};$$

3.4.3 Copy Propagation

Copy Propagation looks for assignments of one variable to another variable like $[\texttt{x:=y}]^l$ in the block l, and for the uses of this variable x in the later blocks.

The Copy Analysis is able to detect the blocks where x is used and will have the same value as y. The transformation replaces these uses of x by the variable y. But this is not the only change performed: there can be unused assignments after that Copy Propagation has replaced the variables, so the unused statements can be removed as well.

Example
The program :

$$[\texttt{u:=a+b}]^1; [\texttt{x:=u}]^2; [\texttt{y:=a*x}]^3$$

becomes

$$[\texttt{u:=a+b}]^1; [\texttt{y:=a*u}]^3$$

after a Copy Propagation transformation.

3.4.4 Dead Code Elimination

The Dead Code Elimination transformation can use the results of one of the two analyses Strongly Live Variables and Live Variables. Both analysis have the same kind of output, so there is no need to write one transformation for each of them. The aim of this transformation is to remove assignments that are not alive at the output of the block they are in i.e., never used again during the rest of the program, whichever path is taken. Whenever the transformation is applied based on the Strongly Live Analysis, assignments of *faint* variables (variables used only in dead or faint assignments) will be removed as well. However, if the block contains a `read` statement, it will not be removed, as the optimization must preserve the fact that the user will be asked for input.

Example
The program :

$$[\texttt{x:=4}]^1; [\texttt{x:=5}]^2; [\texttt{y:=x+5}]^3$$

becomes

$$[\texttt{x:=5}]^1; [\texttt{y:=x+5}]^2$$

after the Dead Code Elimination transformation.

3.4.5 Code Motion

The Code Motion transformation uses the result of the Reaching Definitions Analysis. The aim of this transformation is to move statements that are not influencing the iterations of a loop to a pre-header.
The idea is to find, in a specific loop i.e, *while* loop in the WHILE language used, which statements are loop invariants, and therefore can be move before the loop. In fact, as the condition at the beginning of a *while* loop has to be tested at least once, the new structure will include an *if* statement that tests the condition and after executes the invariants and the *while* loop where the loop invariants have been deleted from the body.

Example
The program :

$$\texttt{while } [\texttt{a=0}]^1 \texttt{ do } [\texttt{i:=j}]^2 \texttt{ od}$$

becomes

$$\texttt{if } [\texttt{a=0}]^3 \texttt{ then } [\texttt{i:=j}]^4; \texttt{while } [\texttt{a=0}]^1 \texttt{ do } [\texttt{skip}]^2 \texttt{ od fi}$$

after a Code Motion transformation.

3.4.6 Elimination with signs

The Elimination with Signs transformation uses the results of the Detection of Signs Analysis to try to predict the value of boolean expressions. This can be useful in the case of boolean expressions being the condition in an *if* statement

or in a *while* loop.

Example
The program :

$$[\texttt{x:=5}]^1; \texttt{while } [\texttt{x>=-9}]^2 \texttt{ do } [\texttt{i:=j}]^3; [\texttt{x:=4}]^4 \texttt{ od}$$

becomes

$$[\texttt{x:=5}]^1; \texttt{while } [\texttt{true}]^2 \texttt{ do } [\texttt{i:=j}]^3; [\texttt{x:=4}]^4 \texttt{ od}$$

after an Elimination with Signs transformation.

This optimization will not often give better results than using a Constant Folding transformation and after evaluating the boolean expression, as most of the time we get result from Elimination with Signs when constant values are used. But we can see that in the case of the example shown above, the value of x could be either 5 or 4, but the Elimination with Signs can still predict the value of the boolean expression, while the Constant Folding will be no use here.

3.4.7 The Precalculation Process

The last transformation performed on the program is not using any analysis previously described. It is in fact a sum of several transformations used to optimize the program. Here are the different operations performed by this process:

1. **Calculation of simple expressions with constants or booleans**: One of the first steps to improve the program is to compute the simple expressions like `4+5` or `true & false`. The compiler has to be able to replace them by their result. Thanks to this, the others parts of the optimization can go further because `4+5` will be the same as `9` and the compiler will be able to tell that `true & false` will always be `false`.
2. **Simplification of an If Statement**: an *if* statement can be simplified if the condition is known to be always `true` or always `false`. If the value of the condition is known, then the *if* statement can be replaced by one

of its bodies. If there is nothing to be executed, like an *if* statement with a `false` condition and without an *else*, the *if* statement is removed.

Example:

$$\texttt{while } [\texttt{x>y}]^1 \texttt{ do if } [\texttt{false}]^2 \texttt{ then } [\texttt{y:=a}]^3 \texttt{ else } [\texttt{y:=b}]^4 \texttt{ fi}; [\texttt{x:=y}]^5 \texttt{ od}$$

becomes

$$\texttt{while } [\texttt{x>y}]^1 \texttt{ do } [\texttt{y:=b}]^4; [\texttt{x:=y}]^5 \texttt{ od}$$

3. **Simplification of an While Statement**: like for an *if* statement, a *while* loop can be simplified if the condition is always `false`: indeed, in this case the *while* statement can be simply deleted.
4. **Removes useless skip statements**: This operation removes all the useless *skip* statements appearing in the program.
5. **Clean the variables Declaration**: Sometimes variables are not used any more in the program. This operation then deletes their declaration in the variable declaration list.

Chapter 4

A new approach for optimizing compilers

This chapter describes a new model for the optimizing compilers. The main concept is to add another module, called the Phase Manager, that will guide the optimization process by deciding what optimization should be done at what time. The first section of this chapter presents the new framework, then the second section introduces the metric-based approach. Finally the third section describes a semi-dynamic ordering using regular expressions.

4.1 Overall framework

In this thesis, a new compiler framework is considered. This framework comprises the three classical components of an optimizing compiler (the frontend, the optimization module and the backend), and another module, called the Phase Manager. The Phase Manager module is the main entity of the whole optimization process. Its goal is to organize the optimization process by using the different approaches described below. It has a direct control over the optimization module and can see the intermediate representation of the program. This allows him to have access to different information concerning the state of the intermediate program during the compilation, and thus it can use these

information when deciding which optimizations should be called.

The remaining of this chapter introduces two different methods the Phase Manager can use to guide the optimization module:

1. **A metric-based approach**: In this approach, the Phase Manager computes after every optimization phases different coefficients, called metrics, that will represent the effect of the different transformations and help him to dynamically choose the following transformation to apply.
2. **Using regular expressions**: This approach allows the user to input a regular expression that will define a specific order of optimization. It also represents a very practical way to perform benchmarking on the optimizing compiler. This technique is fully described in the next section.

The new compiler framework is organized as in Figure 4.1.

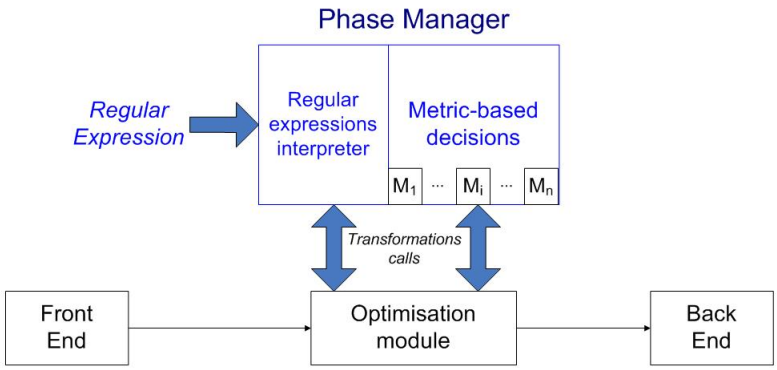


Figure 4.1: Integration of the Phase Manager

4.2 Introduction to the metric-based approach

The main approach considered in this thesis is the metric-based approach. In this approach, the Phase Manager uses the information it can gather about the state of the intermediate representation to decide about the ordering of the optimization phases.

In fact, after each new optimization phase, the Phase Manager calculates some coefficients, called *metrics*, that represent the effects of applying a specific transformation on the program. Once all these metrics have been calculated, the Phase Manager can rank the different optimizations available and choose the most appropriate one that will be apply next. The real interesting point in this approach is that it allows the Phase Manager to have a fully dynamic view of the different effects of the transformations on the intermediate program and thus it can dynamically compute the ordering of the different optimization phases accordingly to the input program.

The metric-based approach is fully detailed in Chapter 5.

4.3 Use of regular expressions

The other approach to the phase-ordering problem considered in this thesis is to introduce a semi-dynamic ordering using regular expressions. These regular expressions will express different ordering patterns to be applied on the intermediate representation of the program. The first part of this section introduces the regular expression framework used. The second part describes the way the regular expressions are interpreted by the Phase Manager. Finally, the last part concerns the creation of a benchmark suite, where a regular expression generator is used to launch a considerable amount of optimizations using regular expressions in order to get some experimental feedback.

4.3.1 Description of the regular expression framework

In this first section, the regular expressions must be defined in a more detailed way. The grammar representing these regular expressions is described in Figure 4.2.

$$
\begin{aligned}
R &\rightarrow R + R \mid R \cdot R \mid R^* \mid (R) \mid I \\
I &\rightarrow \text{CF} \mid \text{DCE} \mid \text{CM} \mid \text{CSE} \mid \text{PRE} \mid \text{CP} \mid \text{ES}
\end{aligned}
$$

Figure 4.2: Grammar for regular expressions

In this grammar, the usual rules of precedence for regular expressions apply. The different identifiers I, which represent the terminals of the regular expression grammar, represent the transformations described in the previous Chapter:

- CF stands for Constant Folding
- DCE stands for Dead Code Elimination
- CM stands for Code Motion
- CSE stands for Common Subexpressions Elimination
- PRE stands for Precalculation
- CP stands for Copy Propagation
- ES stands for Elimination with Signs

As foreseen above, these regular expressions express the order in which different transformations will be applied. The following rules describe the way a regular expression is interpreted by the Phase Manager:

1. For a Union Expression, i.e. $R = R_1 + R_2$, a random choice is made between the two expressions R_1 and R_2 (each expression has 50% of chance to be chosen)
2. For a Concatenate Expression, i.e. $R = R_1 \cdot R_2$, the first expression R_1 is applied, followed by the expression R_2.
3. For a Parenthesis Expression, i.e. $R = (R_1)$, the expression R_1 is applied.
4. For a Star Expression, i.e. $R = R_1^*$, the expression R_1 is applied until no new change is observed by applying R_1 again.
5. For a Symbol Expression, i.e. $R = I$, the specific transformation represented by I is applied.

Though the optimization process is still defined statically by the user choosing the regular expression he wants to use, the use of some type of regular expression, such as Star Expressions (i.e $R = R_1^*$) or Union Expression (i.e $R = R_1 + R_2$) implies dynamic choices and decisions that make the process much more flexible than a sequence of optimization phases statically defined at the compiler construction-time.

4.3.2 Interpretation of the regular expressions

As written above, the Phase Manager is capable, given a specific regular expression, to extract the specific transformations and to order an optimization process by following the different rules described in Section 4.3.1.

This mechanism implemented in the Phase Manager computes the desired order by applying the rules quoted previously. Two interesting rules must be pointed out:

- Rule 1 concerning the Union Expressions: to apply this rule, a number between 0 and 1 is generated randomly, and in the case this number is less than 0.5, the first expression is computed; otherwise the second expression is computed. This way of handling the Union Expression involved a degree of probability, which is by default 50%. This value could also be given as a parameter for the user to decide, or another way of handling this kind of expression could also be addressed: the Phase Manager could perform the two expressions R_1 and R_2 (for $R = R_1 + R_2$) in parallel, and then evaluate the generated program to determine which path is the best optimization between the two of them, and thus take the better choice. However, this raises the issue of performance evaluation, which can take a considerable amount of time if the program has to be executed.

- Rule 4 concerning Star Expressions: to apply this rule, first the actual instance of the program to perform the optimizations on is cloned, and then the expression R_1 (from $R = R_1^*$) is computed. Then the two instances of the program (the current one that has been optimized and the original one) are compared, and if any change has occurred, then the transformation is applied again. In the case the two instances are the same, then the computation of the inner regular expression is stopped. This rule can only be applied under the assumption that it is not possible to apply endlessly the same sequence of optimizations that always performs changes on the program. This assumption is valid for the optimization phases considered in these thesis, but may not be valid for all possible optimizations. Another approach for this rule is to allow the user to specify a maximum number of times the regular expression R_1 should be applied, using an expression like $R = R_1^n$.

Thus this mechanism represents an interesting way of specifying a semi-dynamic order of optimizations, that can evolve depending on how the intermediate representation reacts to the different optimizations. Of course the metric-based

approach should provide a *completely* dynamic ordering that the Phase Manager will used to decide which optimization to perform in real time.

4.3.3 Creation of a benchmark suite

Once this regular expression approach has been implemented in the Phase Manager, a benchmark suite has been designed in order to get some feedback about the effects of different regular expressions, and to be able to compare with the metric-based approach.

4.3.3.1 Structure of the benchmark suite

As explained earlier, the main objective of the benchmark suite is to provide a utility that, coupled with a Regular Expression Generator, allows the user to launch a specified number of regular expressions determining the efficiency of different orders of optimizations on several benchmark programs. Then an analysis is made on the results of these tests in order to determine which regular expressions performed the better on this benchmark programs.

Hence, the benchmarks are composed by:

- The optimization module associated with the Phase Manager
- The Regular Expression Generator providing regular expressions
- Several benchmark programs
- An analyzer, that gets the outputs from the Phase Manager and analyzes results files

The structure of the suite can be observed in Figure 4.3.

The Phase Manager's process that handles the benchmarks contains several steps:

1. It gets the list of benchmark programs

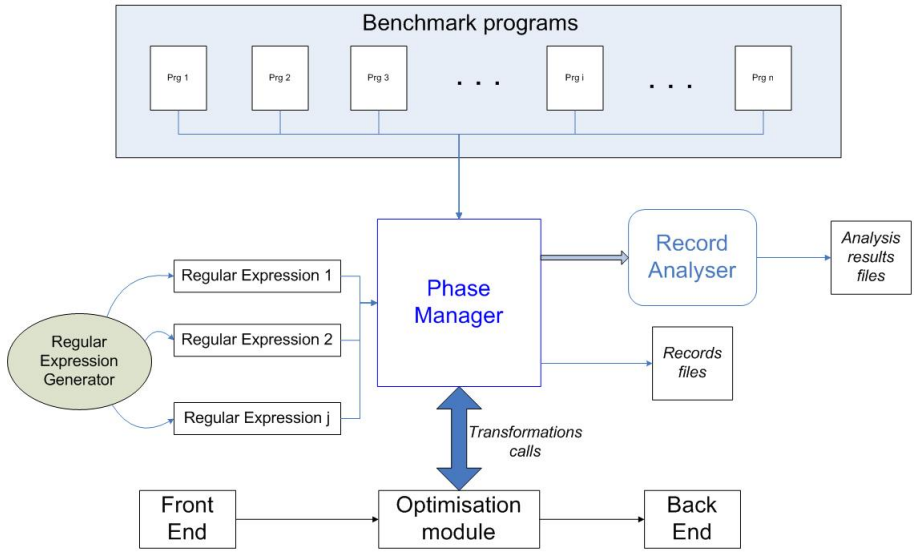


Figure 4.3: Structure of the benchmark suite

2. It generates a specific number of regular expressions using the Regular Expression Generator

3. For each of the pairs benchmark program/regular expression, it applies the regular expression on the program and stores in a record several values relating the optimization, such as the resulting program, the time spent in optimizing, the number of transformation used, etc...(see Section 4.3.3.3)

4. It finally prints out, for each of the benchmark programs, the results stored in the records

Then these results are forwarded to the Record Analyser in order to be analyzed (see Section 4.3.3.4).

4.3.3.2 The Regular Expressions generator

The first module of the benchmark suite is the Regular Expressions Generator. The principle behind this element is the generation of groups of regular expressions that follow different guidelines. Indeed, it consists on different modules generating several different groups of regular expressions.

Firstly, four statically computed regular expressions are added to the final set of regular expressions:

1. $\text{PRE} \cdot (\text{CSE} \cdot \text{CP} \cdot \text{PRE})^* \cdot (\text{CF} \cdot \text{PRE})^* \cdot \text{DCE} \cdot \text{PRE} \cdot \text{CM} \cdot (\text{CF} \cdot \text{ES} \cdot \text{PRE})^*$
2. $\text{PRE} \cdot (\text{CF} \cdot \text{PRE})^* \cdot (\text{CSE} \cdot \text{CP} \cdot \text{PRE})^* \cdot \text{DCE} \cdot \text{PRE} \cdot \text{CM} \cdot (\text{CF} \cdot \text{ES} \cdot \text{PRE})^*$
3. $\text{PRE} \cdot (\text{CSE} \cdot \text{CP} \cdot \text{PRE})^* \cdot (\text{CF} \cdot \text{PRE})^* \cdot (\text{CF} \cdot \text{ES} \cdot \text{PRE})^* \cdot \text{DCE} \cdot \text{PRE} \cdot \text{CM}$
4. $\text{PRE} \cdot (\text{CF} \cdot \text{PRE})^* \cdot (\text{CSE} \cdot \text{CP} \cdot \text{PRE})^* \cdot (\text{CF} \cdot \text{ES} \cdot \text{PRE})^* \cdot \text{DCE} \cdot \text{PRE} \cdot \text{CM}$

These four regular expressions have been designed after the analysis of the dependencies performed further in Section 6.2.2, and represent what could be good order of optimization.

Another module generates n_1 regular expressions containing the concatenation of each transformation only once, in a random order. The result has a probability of 50% to be starred.

A third module generates n_2 regular expressions using static probabilities:

1. Each transformation has a static probability to appear when a Symbol Expression is created. These probabilities has been equally shared between the different transformations, except for Precalculation which has twice more chance to be called, as it is a very cheap transformation (because it does not need any analysis and skims through the program only once):
   - CF: 12.5%
   - CSE:12.5%
   - CP:12.5%
   - DCE:12.5%
   - CM:12.5%
   - ES:12.5%
   - PRE:25%
2. The process of the recursive creation of a new regular expression is as follows:
   - a counter *count* records the number of Symbol Expression already generated. This allows to limit the sequence length to a maximum of 15.

- the method starts by creating a Concatenate Expression then generates its two members R_1 and R_2

- different probabilities are applied when generating members of a regular expression:

  * Generating the members of a Concatenate Expression $R_1 \cdot R_2$:

    ⇒ *For the first member R_1:*
    If *count*<15:
    * either a Symbol Expression (using transformation probabilities described above), a Star Expression or a Union Expression is created.

    else
    * a Symbol Expression is created

    ⇒ *For the second member R_2:*
    If *count*<15
    * either a Concatenate Expression, a Star Expression or a Symbol Expression is created

    else
    * a Symbol Expression is created

  * Generating the members of a Union Expression $R_1 + R_2$:

    ⇒ *For the two members R_1 and R_2, the same process applies:*
    If *count*<15
    * either a Symbol Expression, a Star Expression a Concatenate Expression or a Union Expression is created

    else
    * a Symbol Expression is created

  * Generating the member of a Star Expression R_1^*:

    ⇒ *If the Star Expression is coming from the first member (left child in the parse tree) of a Concatenate Expression:*
    If *count*<15
    * the choice is made randomly between a Symbol Expression, a Concatenate Expression and a Union Expression/

else

* a Symbol Expression is created

⇒ *If the Star Expression is coming from the second member (right child in the parse tree) of a Concatenate Expression:*
If *count*<15

* the Generator chooses between a Symbol Expression, a Concatenate Expression and a Union Expression

else

* a Symbol Expression is created

The fourth module generates n_3 regular expressions using dynamic probabilities for the apparition of the transformation:

1. The method starts with the same probabilities than in the third module (that uses static probabilities).
2. Each time a transformation is used in a Symbol Expression, its probability to appear again is multiplied by a coefficient m (0.5 for example), and the remainder is shared between the other transformations.
3. Then the method uses the same generating process than in the third module to decide the type of regular expressions to be created next.

Finally, the last module generates n_4 regular expressions by concatenating several already computed regular expressions. These expressions have been designed thanks to the analysis of the different interactions between the transformations in Section 6.2.2.The regular expressions to be concatenated are:

1. $(\text{CF} \cdot \text{PRE})^*$
2. $(\text{ES} \cdot \text{PRE})^*$
3. $\text{CSE} \cdot \text{CP} \cdot \text{PRE}$
4. $\text{CP} \cdot \text{CSE} \cdot \text{PRE}$
5. $\text{DCE} \cdot \text{PRE} \cdot \text{CM} \cdot \text{CF} \cdot \text{PRE}$
6. $\text{DCE} \cdot \text{PRE} \cdot \text{CM} \cdot \text{ES} \cdot \text{PRE}$

These expressions are also randomly concatenated using static probabilities.

All the generation of regular expressions is synthesized in the main module of the Regular Expression Generator. It outputs the whole set of regular expressions which can then be used in the benchmark suite.

4.3.3.3 Criteria used to rank the regular expressions

In order to determine which regular expression is describing the best order of optimizations during the benchmark tests, some information concerning the optimization process are gathered during the compilation and stored in a record.

These parameters are:

- The intermediate representation of the optimized program itself.
- The time spent (in milliseconds) during the optimization phase. This value takes into account the whole optimization time, but does not include the time spent in the frontend nor the backend of the compiler.
- A value representing the weighted number of instructions executed in the optimized program. This value is used to compare the performance of the optimized program with the one from the original program and the other version of the optimized program. In order to get this value, the program has to be executed with the Interpreter (Section 3.2.2). The main interest in the weighted number of instructions is that it provides a stable way to compare the programs in performance. Indeed, it would be possible to get the running time of the optimized program, but this running time is always changing depending on how much the machine executing the program is busy, and small changes may not be significant enough to be separated from the overhead due to the start of a background process on the executing machine.
- The number of transformations used when optimizing the program. Indeed, each call to a transformation is recorded: the less transformations are used, the better is the order of optimizations.
- The number of Skip statements and variable declarations occurring in the optimized program. Skip statements and variable declarations are not changing anything in the behavior of the optimized program, but it should be removed as much as possible when useless, so the program is smaller and unused variables does not have any allocated space in the memory.

As explained before, this records are, after having been created by the Phase Manager during the benchmark runs, transferred to the Record Analyser that will use them to rank the different regular expressions. All the parameters recorded in these objects are used as criteria to evaluate the regular expressions, though some have more importance than others. With a goal of optimization set to the shortest running time of the program, the weighted number of executed instructions is of primary importance, as it evaluates the degree of optimization of the program, and before improving the speed of the compiler, it must be ensured that the program is optimized the better. After this, the time spent while optimizing and the number of transformations used are also very important to determine which order is the best one, whereas the number of Skip statements and variable declarations is not fundamental, though it can still interesting.

Thanks to these parameters, a general ranking can be made by first choosing the regular expression giving a program with the least number of instructions, then with the smallest optimization time, number of transformations and finally number of skip statements and variable declarations. In case the goal of the optimization process may change from speed to program size, the importance of the parameters may change as well, as removing useless skip statements and variable declaration may be much more profitable.

4.3.3.4 Analysis: the Record Analyser

Once the benchmark tests have been performed, the results stored in the records are transfered to the Record Analyser. This Analyser gets all the data from these records to compute different rankings and outputs the result.

This module is composed by three components:

- The first two ones contains five rankings:
    1. A ranking on the time spent by a regular expression while optimizing.
    2. A ranking on the number of executed instructions in the optimized program.
    3. A ranking on the number of transformations used in the optimization process.
    4. A ranking on the number of skip statements and variable declaration in the optimized program.
    5. A general ranking of all the regular expressions, using the previous rankings and the order of importance of each parameter to globally rank the regular expressions.

- The third component can record the "points" earned by a regular expression. Indeed, each time a regular expression is in a ranking, it earns a specific number of points in order to be globally ranked in the whole benchmark general ranking.

Figure 4.4 shows the overall structure of the analyzer.

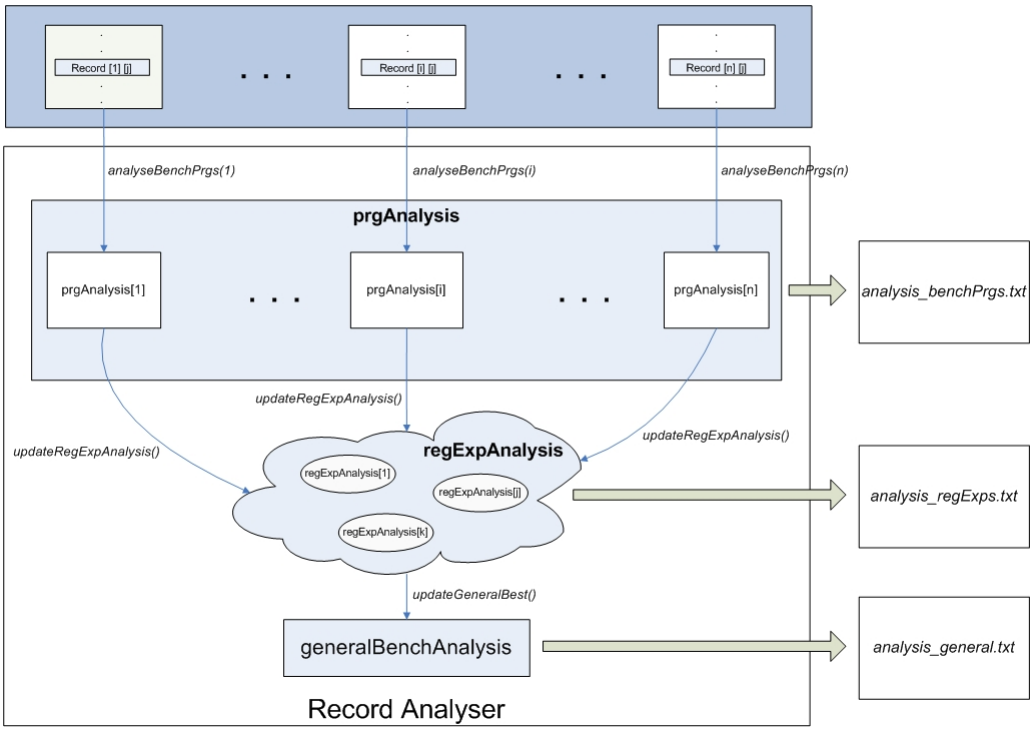


Figure 4.4: Structure of the analyzing process: the regExpAnalysis is the third component that globally ranks the regular expressions, while the two others (prgAnalysis and generalBenchAnalysis) have the five rankings and rank either each program or the whole benchmark.

4.3.3.5 Results of the benchmarks

Once the benchmark suite has been set up, tests have been made in order to get some insights about how applying different orders of optimizations (through different regular expressions) can affect the degree of optimization of the pro-

gram.

These tests have been done by generating 500 regular expressions. This number have been arbitrarily chosen. More regular expressions could have been generated but it would have induced a considerable increase in the amount of time needed for the benchmarks. Figure 4.5 represents the results for the best regular expression for each benchmark program. To choose the best regular expression, the algorithm:

1. takes all the regular expressions with the lowest number of instructions executed.
2. among them takes the regular expressions spending the least time in optimization.
3. sorts the remaining regular expressions according to the number of transformation used, and finally the number of skip and variable declarations.

	Weighted number of instructions exec.	Transformations applied	Time spent (in ms)
BenchPrg1	896	6	491
BenchPrg2	3708603	4	2770
BenchPrg3	171925	11	2120
BenchPrg4	121216	2	197
BenchPrg5	97247	0	27
BenchPrg6	51710	3	219
BenchPrg7	431514	0	25
BenchPrg8	1590652	9	721
BenchPrg9	3508	30	256
BenchPrg10	39826115	3	10482
BenchPrg11	26407	36	994
BenchPrg12	1705	8	53
BenchPrg13	1108	10	164
BenchPrg14	258	0	157
BenchPrg15	498190	24	1487
BenchPrg16	14109	36	532
BenchPrg17	34096	30	1408

Figure 4.5: Results for the best regular expression in the benchmark suite

These results represents some of the best optimization runs that could be performed on the benchmark programs used. Hence, it provides a good basis to be compared with the results from the metric-based approach defined in the next chapter (the comparison is made in Section 6.1).

It could also be interesting to focus on the regular expressions that performed well in these benchmarks. The following graph (Figure 4.6) gives the percentage of regular expressions where the number of instructions executed is at the optimum.

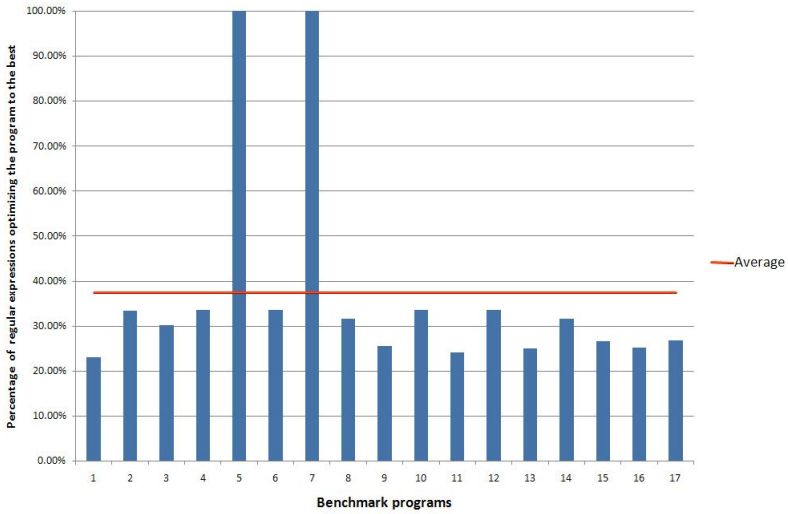


Figure 4.6: Percentage of regular expressions reaching the minimum number of instructions executed

This figure shows that for most of the benchmark programs, more than half of the regular expressions optimize the program less than the optimum observed. The two benchmark programs (BenchPrg 5 and 7), where 100% of the regular expressions produce the best code, are the two programs that cannot be optimized at all (the best regular expressions is the empty one). Assuming that the best regular expression produces an optimal program, it would mean that more than half of the regular expressions are not producing a program optimized enough.

Another interesting point is to evaluate the regular expressions for the whole benchmark. The results from the Record Analyser show that there is no regular expression that clearly outperforms the rest. Indeed, on these benchmarks, a group of regular expressions share the top of the general ranking, and most of the regular expressions in at least the 25 first places have been observed to be one of the best regular expressions for at least a benchmark program.

Finally, these benchmarks have highlighted several interesting points:

- It is very difficult (verily impossible) to design an regular expression that would compute an optimal order of optimization for most of the programs. Thus, the use of statically defined optimization sequences is very unlikely to produce the best code for most of the programs.
- Nevertheless, allowing the user to provide his own regular expression for the program he wants to optimize could be an interesting step towards a better optimization process. However, this would require that the user has sufficient knowledge about the optimization phases available and that it is able to design a good regular expression for each of his specific programs.
- Finally, the best regular expressions can still be good indicators of how much the different programs can be optimized, and it can be very interesting to compare these results with the ones coming from the metric-based approach.

The conclusion of these tests is that this approach, however interesting for benchmarking, is not very suited for practical compiling. It can probably take less time that the common iterative compilation process, but ensuring a good probability of having an optimal program at the end requires to generate and apply a considerable amount of regular expressions, which can become very long. Therefore, a more advanced compilation process which will be dynamic and relatively fast is clearly necessary.

Chapter 5

Phase-ordering using a Metric-based Approach

In the previous chapter, an approach based on regular expressions has been considered. The main approach to the phase-ordering problem developed in this thesis is another one based on coefficients, called metrics, to evaluate the potential of optimization a specific transformation can have on the program. During the optimization process, these metrics can then be all compared in order to approximate how useful the available transformations will be and apply the one that looks the most adequate. Thus, the state of the intermediate representation is taken into account. This chapter relates how this approach has been defined: the first two parts give an overview of the approach, while the third and fourth part contain detailed descriptions of the different metrics; finally the last part deals with the algorithm implemented to use the metrics' values to perform the phase-ordering.

5.1 The metric-based approach

The main idea is to develop a dynamic ordering of transformations, depending on the different states the program goes through during the optimization process. Thus, the metrics should take into account the different points of the program

where the corresponding transformation will be useful, in order to represent as much as possible the interest the Phase Manager can have in performing a specific transformation instead of another.

Thus, the different metrics will approximate the different analyses used to perform the corresponding transformations, and then guess the number of blocks in the program where each transformation would actually be applicable. However, even though the main aim of this approach is to generate a dynamic order of transformations applied to a specific program, an important parameter to take into account is the time spent while optimizing, as an implicit aim is to generate this order while decreasing the average optimization time, or at least not increasing it. That is why it appears totally impossible to use the MFP (Maximal Fixed Point) or an abstract worklist algorithm (described in Section 2.2.3) to compute all the analyses and then use these results to get the metrics.

5.2 Choice of the different metrics

When choosing a metric it is important to make sure that it:

1. has a complexity that is at least an order of magnitude smaller than that of the analysis, so that the calculation of the metric will take much less time than performing the real analysis
2. has a non-zero value whenever a transformation can be used to optimize the program
3. never decreases whenever the use of the associated transformation becomes more interesting, in terms of number of changes that can be performed in the program

The *first* point is of a great importance in the metric definitions. Indeed, the metrics' complexity must be relatively small not to increase the compilation time too much. Then the use of the different algorithms appearing in [16] to compute the analysis, such as the MFP, MOP, or worklist algorithm, is to be avoided, as it uses the flow graph to calculate the entry and exit informations of the analysis, with a non-linear time whenever the program to optimize contains loops. Instead of these algorithms, an algorithm that will be called *propagation algorithm* will be used to approximate these analysis information in linear time. This algorithm is detailed in Section 5.3.2.1.
This issue also has another consequence on the choice of the different metrics.

This concerns more precisely the Precalculation Process: this transformation is different from the other ones, as it does not use any analysis information, but just goes through the program once and makes different changes, as shown in Section 3.4.7. Hence this transformation is much cheaper than the other ones, and much faster. That is why this transformation will be applied once after every other transformations. In Section 6.2.2, this will be compared to another approach, where a metric for the Precalculation Process will be used.

The *second* point also uncovers an interesting issue. In fact, the metrics aims at helping the Phase Manager to rank all the available transformations, and determine which one should be applied next on the program. In order to perform the necessary amount of transformations to get a program as optimized as possible at the end of the process, this second rule is very important. Thus, for the metrics to respect this rule, and as a result of the first rule as well, the metrics should compute an over-approximation: this may lead to unnecessary transformations, but it will ensure that the program will be optimized as best as possible, which is the most important criteria.

Finally, the *third* point is just used to make sure that the parameters involved in a metric computation should be related to the parameters used during the call of a transformation associated with the metric. For example, the parameters used in the metric corresponding to *Common Subexpression Elimination* should consider expressions and their occurrences in the program, because it is the most important factor that is concerned by this optimization.

As a result of these rules, a metric will be defined for each of the transformations available in the compiler.

5.3 Approximation of the different analyses

In order to calculate the metrics' values, the different analyses involved in the transformations represented by all of these metrics have to be approximated. As explained in the previous section, it is totally impossible to use the results coming from MFP or MOP to get these information, as it would significantly increase the compilation time. So instead of using these algorithms, which give exact analysis results, an algorithm called *propagation algorithm* has been implemented to get an over-approximation of the analysis information.

This algorithm visits nodes in a linear way, and thus reduces the computation time, while of course reducing the analysis accuracy. In the following sections the propagation functions associated with all used analyses are described, as

well as the general propagation algorithm.

5.3.1 Visiting graph

When presenting the different propagation functions and the general propagation algorithm, different functions will be used, creating a visiting graph from the program.

Initial and final labels. The first function is $initV$: **Stmt** $\rightarrow$ **Lab**.

This function returns the *initial label* of a statement in a visiting graph, and is very similar to the *init()* function defined in [16] for the flow graph of the WHILE language:

$$\begin{aligned}
initV([x := e]^l) &= l \\
initV([\text{skip}]^l) &= l \\
initV(\text{write } [e]^l) &= l \\
initV([\text{read } x]^l) &= l \\
initV(S_1; S_2) &= initV(S_1) \\
initV(\text{begin } D\ S \text{ end}) &= initV(S) \\
initV(\text{if } [e]^l \text{ then } S \text{ fi}) &= l \\
initV(\text{if } [e]^l \text{ then } S_1 \text{ else } S_2 \text{ fi}) &= l \\
initV(\text{while } [e]^l \text{ do } S \text{ od}) &= l
\end{aligned}$$

Similarly, function $finalV$: **Stmt** $\rightarrow$ **Lab** returns the single *final label* for each statement.

It is defined by:

$$\begin{aligned}
finalV([x := e]^l) &= l \\
finalV([\text{skip}]^l) &= l \\
finalV(\text{write } [e]^l) &= l \\
finalV([\text{read } x]^l) &= l \\
finalV(S_1; S_2) &= finalV(S_2) \\
finalV(\text{begin } D\ S \text{ end}) &= finalV(S) \\
finalV(\text{if } [e]^l \text{ then } S \text{ fi}) &= finalV(S) \\
finalV(\text{if } [e]^l \text{ then } S_1 \text{ else } S_2 \text{ fi}) &= finalV(S_2) \\
finalV(\text{while } [e]^l \text{ do } S \text{ od}) &= l
\end{aligned}$$

Blocks and labels. The visiting graph uses two functions defined in [16], which represent the mapping of statements with the blocks of the graph, as well as the set of labels occurring in the statements:

$blocks$: **Stmt** $\rightarrow \mathcal{P}$(**Blocks**)
$labels$: **Stmt** $\rightarrow \mathcal{P}$(**Lab**)

Visit and reverse visit functions. These functions allow the construction of the visiting graph by defining its edges, or *visiting flows*, using the function $visit$: **Stmt** $\rightarrow \mathcal{P}$(**Lab** $\times$ **Lab**).

This function maps statement to sets of visiting flows:

$$\begin{aligned}
visit(S_1; S_2) &= visit(S_1) \cup visit(S_2) \\
&\quad \cup \{(finalV(S_1), initV(S_2))\} \\
visit(\text{begin } D\ S \text{ end}) &= visit(S) \\
visit(\text{if } [e]^l \text{ then } S \text{ fi}) &= visit(S) \cup \{(l, initV(S))\} \\
visit(\text{if } [e]^l \text{ then } S_1 \text{ else } S_2 \text{ fi}) &= visit(S_1) \cup \{(l, initV(S_1))\} \cup visit(S_2) \\
&\quad \cup \{(finalV(S_1), initV(S_2))\} \\
visit(\text{while } [e]^l \text{ do } S \text{ od}) &= visit(S) \cup \{(l, initV(S)), (finalV(S), l)\} \\
visit(B^l) &= \emptyset \text{ otherwise}
\end{aligned}$$

For backward analyses, the reverse function $visit^R$: **Stmt** $\rightarrow \mathcal{P}$(**Lab** $\times$ **Lab**) is used. This function is defined by: $visit^R(S) = \{(l, l') | (l', l) \in visit(S)\}$

Figure 5.1 shows an example of a visiting graph for a very simple program based on the example from Figure 3.3. It has to be pointed out that the visiting graph is only used for creating a visiting order while using the *propagation algorithm*, and in no case is an equivalent for the flow graph.

Two other functions are defined on the visiting graph:

- $nextVisit$ **Lab** $\rightarrow \mathcal{P}$(**Lab**), which, given a label, returns all the next elements of the visiting graph for this specific label. The set of next elements is in fact a singleton for all blocks except if the block is a loop condition, in which case there are two elements (one going in the loop's body and one leaving the loop).

- $nextVisit^R$ **Lab** $\rightarrow \mathcal{P}$(**Lab**), which does the same for the reverse visiting graph.

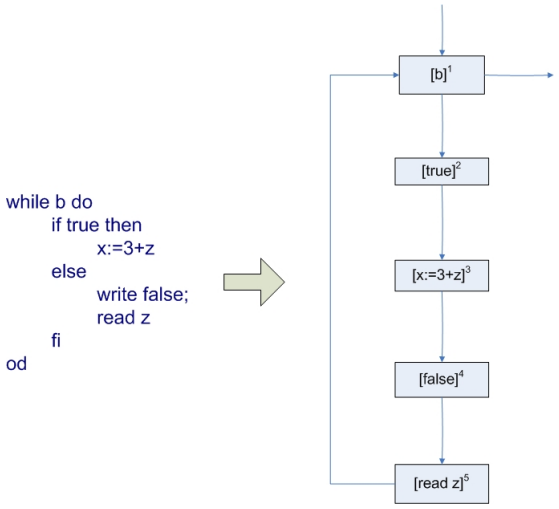


Figure 5.1: Example of a visiting graph

5.3.2 Analyses of Bit-Vector Framework

This section establishes, for all the analyses of the bit-vector framework implemented in the WHILE compiler, the different parameters and functions used in the *propagation algorithm*, as well as the algorithm itself.

The propagation algorithm uses *kill* and *gen* functions for bit-vector analyses, but as the lattices are changing, these functions must be re-defined.

These new functions are also used in a transition function, which is the same as the one defined for classical algorithms:

$$f^{approx}(\lambda) = (\lambda \backslash kill^{approx}(B^l)) \cup gen^{approx}(B^l), \text{ where } B^l \in blocks(S_*)$$

5.3.2.1 General propagation mechanism

The *propagation algorithm* follows a simple mechanism that permits to approximate the different analyses the transformations need. Two main functions are used in this mechanism: a *transition* function and a *propagation* function.

The *transition* function is proper to each of the analyses, and defines the way to modify the data when computing the analyses on a specific block, using *kill* and *gen* functions. These functions will be defined in the following sections.

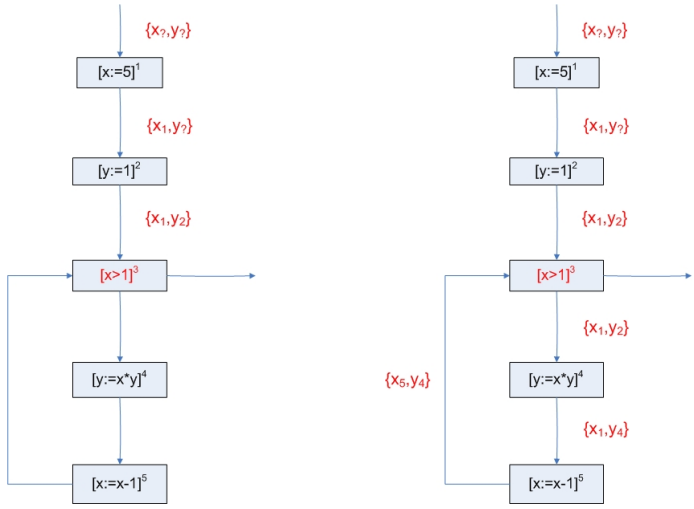
For each of the analyses, a *propagation* function must be described as well. This propagation function is used when the computation comes back to a node already visited, e.g. the condition of a loop. It is then used to propagate new information to the entry and exit points of all the labels within the loop, according to the corresponding least upper bound associated with the analysis.

Consider the Reaching Definitions Analysis as an example, with the following code:

$$[\texttt{x:=5}]^1;[\texttt{y:=1}]^2;\texttt{while } [\texttt{x>1}]^3 \texttt{ do } [\texttt{y:=x*y}]^4;[\texttt{x:=x-1}]^5 \texttt{ od}$$

This analysis is chosen as an example, because the different parameters of the lattice and the *transition* function are the same as in the classical analysis, as can be seen in the next section.

Figure 5.2 shows the visiting graph and the exit and entry before and after having been in the loop, but still before propagation. This first step is simply



(a) Before entering the loop

(b) After the loop

Figure 5.2: Visiting graph before propagation

updating the entry and exit informations of all labels. The use of the visiting graph makes this step easy and accurate, except for loops, but that is where the propagation must be made. Note that in this step if statements do not raise

any issues, as their bodies are treated sequentially, based on the visiting graph.

Once this first part is computed, the data must then be propagated through the loop. The two entry information of the loop condition, respectively called *entryUp* for the one coming from the body of the loop and *entryDown* for the one coming from the blocks before the loop, must be compared, for each variable. Then, if any information are present in the *entryUp* information, it must be propagated to all the entry/exit informations that the data from *entryDown* are able to reach. This propagation mechanism for variable x can be seen in Figure 5.3. In this particular example, the informations must be *added* to the entry/exit information of the loops.

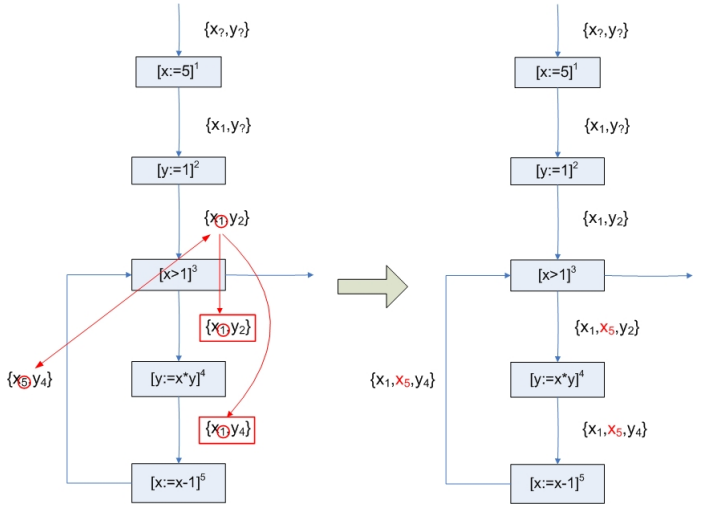


Figure 5.3: Propagation of variable x

Two particular issues must be pointed out. The first one concerns the lattices of the analyses. The important part of this propagation is to know where the data must be propagated. In the case of the Reaching Definitions Analysis, it is easy to follow the data from the *entryDown* information until when it is killed. In the case of other analyses, lattices must often be adapted to keep track of where the data are defined/used: for the Available Expression analysis (Section 5.3.2.4), the lattice must be changed to $\mathcal{P}(\mathbf{AExp}_* \times \mathbf{Lab}_*^?)$ so it is possible to follow the expressions defined at the entry of a loop and know until where to propagate (or kill in this case) the information.

Another issue is the occurrence of nested inner loops. To solve this problem, it is necessary to keep track of the level of nesting of each block in the loop, and

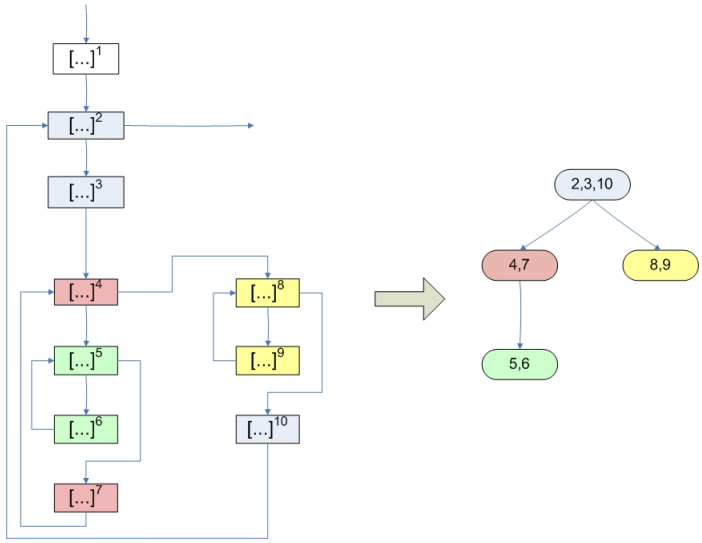


Figure 5.4: Nested loops and propagation tree

whenever a propagation is made at one loop's entry, the data is also propagated to its *children*. An example of nested loops and of the corresponding propagation tree can be seen in Figure 5.4.

The propagation algorithm, as described in Algorithm 1, has been implemented in the Phase Manager. In this algorithm, the functions *blocks()* and *labels()* defined in Section 5.3.1 are used to determine if a block is a loop condition, and if so, the blocks from the loop's body are first evaluated, then the propagation is made and finally the algorithm leaves the loop.

As explained in Section 5.3.1, the set of next elements of a block in the visiting graph is a singleton for every block except the one corresponding to a loop's condition. Moreover, in the case of the block being a loop's condition, the nested loops propagation tree has to be updated, thanks to the **PropTreeNode** object (which represents a node in the propagation tree, as in Figure 5.4).

```
STEP 1:INITIALIZATION:
W = ∅ ; // W = the list of visited labels
foreach l' in labels(S*) do
  | if l = E then approx∘(l) = ι;
end
call iterationStep(E,null);

STEP 2:METHOD iterationStep(Lab curLabel,PropTreeNode ptn):
if curLabel ∉ W then
  | W = insert(curLabel);
  | previousLabs = the predecessors of curLabel in the flow graph
  | approx∘(curLabel) = ⨆_{l'∈W∩previousLabs} approx•(l');
  | approx•(curLabel) = f_l^approx(approx∘(curLabel));
  | if B^curLabel ≠ loop condition then
  |   | if ptn ≠ null then add curLabel to ptn;
  |   | nextLab = get the single next element of curLabel in the visiting graph;
  |   | call iterationStep(nextLab,ptn);
  | else
  |   | newPtn = create new PropTreeNode;
  |   | add newPtn to the sons of ptn;
  |   | entryDown = approx∘(curLabel);
  |   | nextLabels = next labels of curLabel in visiting graph;
  |   | labelsInBody = labels(loop's body statement);
  |   | foreach l in nextLabels do
  |   |   | if l ∈ labelsInBody then call iterationStep(l,newPtn);
  |   | end
  |   | entryUp = ⨆_{l'∈W∩labelsInBody} approx•(l');
  |   | compare entryUp and entryDown and propagate entry/exit
  |   | information to newPtn and its sons;
  |   | foreach l in nextLabels do
  |   |   | if l ∉ labelsInBody then call iterationStep(l,ptn);
  |   | end
  | end
end
```

Algorithm 1: Propagation algorithm for bit-vector framework analyses

5.3.2.2 Reaching Definition Analysis

The parameters used in the propagation algorithm to approximate the different analyses are now defined, starting by the Reaching Definitions Analysis. As pointed out in the previous section, the metric for Reaching Definitions uses the same parameters as the analysis, because the lattice $\mathcal{P}(\mathbf{Var}_* \times (\mathbf{Lab}_*^?))$

allows already to use traceable information. Of course, the visiting graph corresponding to a forward analysis must still be added. Moreover, due to the use of the visiting graph, the parameter E now represents a single extremal label, for the Reaching Definitions Analysis and all the analyses from the bit-vector framework as well.

Here is a recall of the parameter used for the Reaching Definition Analysis:

- $\mathcal{P}(\mathbf{Var}_* \times (\mathbf{Lab}_*^?))$, the lattice of variables and labels, indicating where a variable is last defined
- a partial ordering $\subseteq$ and least upper bound operation $\bigcup$ that shows we have a *may*-analysis
- $\{(x, ?) \mid x \in FV(S_*)\}$, the extremal value ι of the analysis
- $init(S_*)$, the single extremal label E
- $flow(S_*)$, the definition of the flow F through the program
- $visit(S_*)$, the forward visiting graph
- the transition function f^{approx}

For the same reasons, the $kill^{approx}$ and gen^{approx} functions remain the same (see Section 3.3.1).

5.3.2.3 UD and DU chains

With the propagation algorithm applied to the **Reaching Definitions Analysis**, it is then possible to obtain the Use-Definition and Definition-Use chains. These chains are defined in a similar way as in Section 3.3.2, by replacing $\mathrm{RD}_\circ(l)$ by the result of the propagation algorithm for Reaching Definitions Analysis, $approx_\circ^{\mathrm{RD}}(l)$.

5.3.2.4 Available Expressions Analysis

When looking at the classic parameters for the Available Expressions Analysis, it can be observed that the lattice deals only with expressions, with no way to distinguish two instances of the same expression used at different places. This raises an issue when propagating, since it may occur that two instances of the

same expression are defined in the same loop, and only one of them must be eliminated because of the propagation, as can be seen in Figure 5.5.

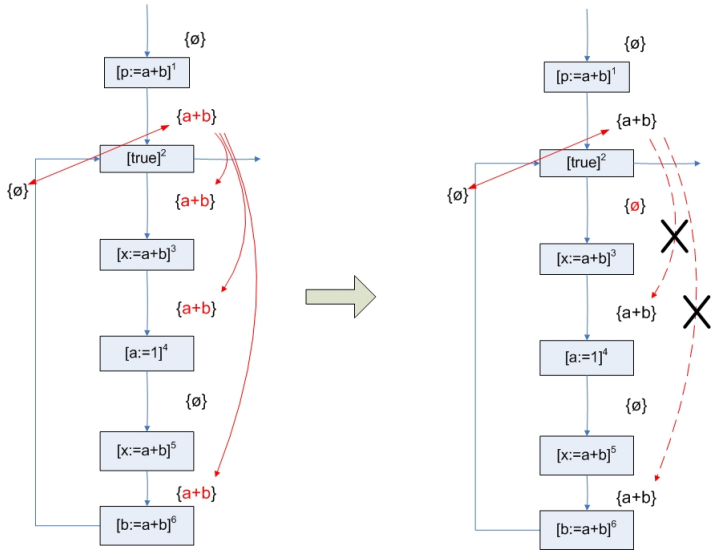


Figure 5.5: Issue with Available Expression lattice

That is why in the lattice, for each instance of an expression, expressions are now stored with the label where they are used. The parameters are then:

- $\mathcal{P}(\mathbf{AExp}_* \times \mathbf{Lab}_*^?)$, the lattice of all pairs containing non-trivial arithmetic expressions occurring in the program and the label of their use
- a partial ordering $\supseteq$ and least upper bound operation $\bigcap$ that shows we have a *must*-analysis
- $\emptyset$, the extremal value ι of the analysis
- $initV(S_*)$, the single extremal label E
- $flow(S_*)$, the definition of the flow F through the program
- $visit(S_*)$, the forward visiting graph
- the transition function f^{approx}

As the lattice has been changed, the $kill^{approx}$ and gen^{approx} functions also need to be changed:

$$
\begin{aligned}
kill_{AE}^{approx}([x := a]^l) &= \{(a', l') \mid a' \in \mathbf{AExp}_* \wedge x \in FV(a') \wedge l' \in \mathbf{Lab}_*^?\} \\
kill_{AE}^{approx}([\text{read } x]^l) &= \{(a', l') \mid a' \in \mathbf{AExp}_* \wedge x \in FV(a') \wedge l' \in \mathbf{Lab}_*^?\} \\
kill_{AE}^{approx}([b]^l) &= \emptyset \\
kill_{AE}^{approx}([skip]^l) &= \emptyset \\
gen_{AE}^{approx}([x := a]^l) &= \{(a', l) \mid a' \in \mathbf{AExp}(a) \wedge x \notin FV(a')\} \\
gen_{AE}^{approx}([b]^l) &= \{(b', l) \mid b' \in \mathbf{AExp}(b)\} \\
gen_{AE}^{approx}([\text{read } x]^l) &= \emptyset \\
gen_{AE}^{approx}([skip]^l) &= \emptyset
\end{aligned}
$$

Using this new lattice, the propagation mechanism gives correct results, as can be seen in Figure 5.6. The labeling of the expressions permits to follow the ones that were at the beginning of the loop and eliminate them if needed in the loop.

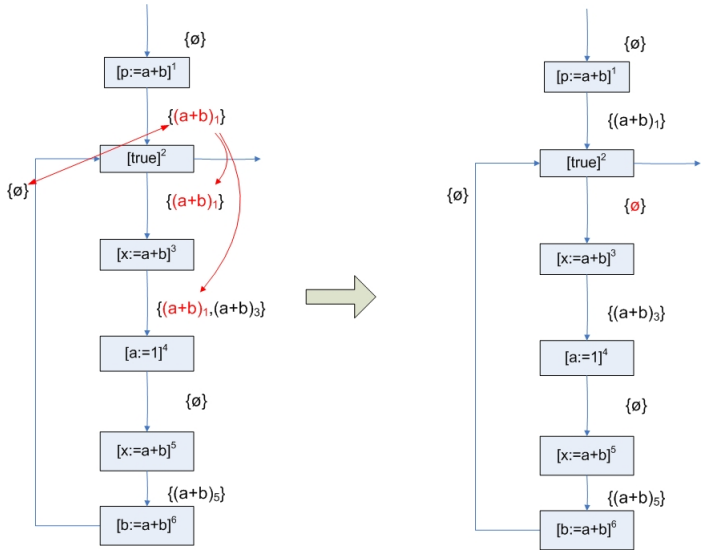


Figure 5.6: Available Expression Analysis propagation with new lattice

5.3.2.5 Copy Analysis

The parameters for the Copy Analysis raise the same issue as with the Available Expressions Analysis, which also causes the need of labeling the different pairs of copy variables for propagation need. Thus the parameters are:

- $\mathcal{P}(\mathbf{Var}_* \times \mathbf{Var}_* \times \mathbf{Lab}_*^?)$, the lattice of all triples of two variables and a label
- a partial ordering $\supseteq$ and least upper bound operation $\bigcap$ that shows we have a *must*-analysis
- $\emptyset$, the extremal value ι of the analysis
- $initV(S_*)$, the single extremal label E
- $flow(S_*)$, the definition of the flow F through the program
- $visit(S_*)$, the forward visiting graph
- the transition function f^{approx}

The definitions of the $kill^{approx}$ and gen^{approx} functions used in the transition function f_l^{approx} are:

$$\begin{aligned}
kill_{CA}^{approx}([x := a]^l) &= \{(x, y, l'), (y, x, l') | y \in FV(S^*) \wedge l' \in \mathbf{Lab}_*^?\} \\
kill_{CA}^{approx}([\text{read } x]^l) &= \{(x, y, l'), (y, x, l') | y \in FV(S^*) \wedge l' \in \mathbf{Lab}_*^?\} \\
gen_{CA}^{approx}([x := a]^l) &= \{(x, y, l) | a = y \wedge y \in FV(S^*)\} \\
kill_{CA}^{approx}(B^l) &= gen_{CA}^{approx}(B^l) = \emptyset \quad \text{otherwise}
\end{aligned}$$

5.3.2.6 Very Busy Expressions Analysis

The parameters for the Very Busy Expressions Analysis follow closely the same patterns as the Available Expressions Analysis, except that it is a backward analysis. These parameters are:

- $\mathcal{P}(\mathbf{AExp_*} \times \mathbf{Lab}_*^?)$, the lattice of all pairs containing non-trivial arithmetic expressions occurring in the program and labels
- a partial ordering $\supseteq$ and least upper bound operation $\bigcap$
- $\emptyset$, the extremal value ι of the analysis

- $finalV(S_*)$, the single extremal label E
- $flow^R(S_*)$, the definition of the flow F through the program
- $visit^R(S_*)$, the forward visiting graph
- the transition function f^{approx}

The definitions of the $kill^{approx}$ and gen^{approx} functions are:

$$\begin{aligned}
kill_{VB}^{approx}([x := a]^l) &= \{(a', l') \mid a' \in \mathbf{AExp_*} \wedge x \in FV(a') \wedge l' \in \mathbf{Lab}_*^?\} \\
kill_{VB}^{approx}([\text{read } x]^l) &= \{(a', l') \mid a' \in \mathbf{AExp_*} \wedge x \in FV(a') \wedge l' \in \mathbf{Lab}_*^?\} \\
kill_{VB}^{approx}([b]^l) &= \emptyset \\
kill_{VB}^{approx}([skip]^l) &= \emptyset \\
gen_{AE}^{approx}([x := a]^l) &= \{(a', l) \mid a' \in \mathbf{AExp}(a)\} \\
gen_{AE}^{approx}([b]^l) &= \{(b', l) \mid b' \in \mathbf{AExp}(b)\} \\
gen_{VB}^{approx}([\text{read } x]^l) &= \emptyset \\
gen_{VB}^{approx}([skip]^l) &= \emptyset
\end{aligned}$$

5.3.2.7 Live Variables Analysis

The parameters for the last analysis of the Bit Vector Framework, the Live Variables Analysis, are also different from the classic parameters seen in Section 3.3.6. Again, the main issue is to know until where to propagate the data. That is why this analysis uses the lattice $\mathcal{P}(\mathbf{Var}_* \times \mathbf{Lab}_*^{+/-})$, where $\mathbf{Lab}_*^{+/-} = \mathbf{Lab}_* \cup \{0\} \cup \{0 - l | l \in \mathbf{Lab}_*\}$. The idea is that all variables should be present in the entry and exit sets, but the ones that are dead will be those without a strictly positive value for the label. Each time a variable is declared dead, its label is set to the next negative label. A single value for dead variable cannot be used, as there can be several kills of the same variable inside a loop, and the propagation must be able to see the difference. This mechanism is illustrated in Figure 5.7.

The mechanism has to use a general counter $next_{dead}$ to recall the value of the last negative label given to the last dead variable. The parameters for the lattice are:

- $\mathcal{P}(\mathbf{Var}_* \times \mathbf{Lab}_*^{+/-})$, the lattice of all variables occurring in the program with an integer label
- a partial ordering $\subseteq$ and least upper bound operation $\bigcup$ that shows we have a *may*-analysis

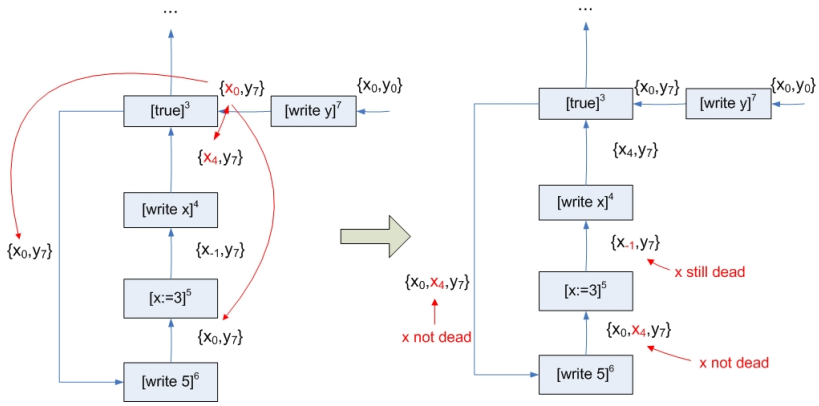


Figure 5.7: Mechanism for Live Variable Analysis

- $\{(x, 0) \mid x \in FV(S_*)\}$, the extremal value ι of the analysis
- $final(S_*)$, the single extremal label E
- $flow^R(S_*)$, the definition of the flow F through the program
- $visit^R(S_*)$, the forward visiting graph
- the transition function f^{approx}

The definitions of the $kill^{approx}$ and gen^{approx} functions are:

$$
\begin{aligned}
kill_{LV}^{approx}([x := a]^l) &= \{(x, l') | l' \in \mathbf{Lab}_*^{+/-}\} \\
kill_{LV}^{approx}([\text{read } x]^l) &= \{(x, l') | l' \in \mathbf{Lab}_*^{+/-}\} \\
kill_{LV}^{approx}([b]^l) &= \emptyset \\
kill_{LV}^{approx}([skip]^l) &= \emptyset \\
gen_{LV}^{approx}([x := a]^l) &= \{(y, l) | y \in FV(a)\} \cup \{(x, next_{dead}), \text{ if } x \notin FV(a)\} \\
gen_{LV}^{approx}([b]^l) &= \{(y, l) | y \in FV(b)\} \\
gen_{LV}^{approx}([\text{read } x]^l) &= \{(x, next_{dead})\} \\
gen_{LV}^{approx}([skip]^l) &= \emptyset
\end{aligned}
$$

Note that each time the counter $next_{dead}$ is assigned to a variable, it is decremented. This counter is in $\{0\} \cup \{0 - l | l \in \mathbf{Lab}_*\}$ because it is decremented once at each assignment or read statement: there are at most n blocks like this in the program, so it is less than the total number of blocks (and then less than the total number of labels): $n \leq n_{blocks}$. This is important because it makes sure that $next_{dead}$ is always within the bounds of the lattice $\mathcal{P}(\mathbf{Var}_* \times \mathbf{Lab}_*^{+/-})$.

5.3.3 Constant Propagation Analysis

The propagation algorithm defined in Section 5.3.2 can be modified in order to compute the approximation of some analyses which are based neither on the bit-vector framework nor on the distributive framework. This is, for example, the case for the Constant Propagation Analysis.

The idea is to use the propagation algorithm with the parameters for the Reaching Definitions Analysis, and to add a mapping that will, for each variable definition, contain the value of the variable ("?" if not constant) and link this variable definition to all the variables that are used in the evaluation of this value. This will then create some constraints that can be used after to add the propagation mechanism. This mapping function is called the σ function:

$$\sigma : (\mathbf{Var} \times \mathbf{Lab}) \rightarrow (\mathcal{P}(\mathbf{Var} \times \mathbf{Lab}) \times \mathbf{Value})$$

where the **Value** can be either a boolean or an integer. The parameter $\mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$ contains the set of reaching definitions of the variable used in the evaluation of the variable's value.

Figure 5.8 shows an example of how the σ function is used. It represents the small program:

```
x:=1;y:=5;if true then (if y>1 then y:=2*x-1 else write x fi;
y:=x+1) fi;write y
```

The σ function stores, from the current reaching definitions information available, the possible reaching definitions of the variables used in the current variable definition, and computes its value according to this set (each value is computed and the result is a single value if they are all the same, ? otherwise).

During propagation, the sets of reaching definitions are updated like for Reaching Definitions Analysis, and the value is updated as well. In fact, if the values of the different reaching definitions are not the same, a "?" has to be propagated where the value were constant before. To perform this propagation, the process starts from the σ node of the reference node (coming from the *entryDown* set) and follows the links of used variables backward. An example of this propagation mechanism can be seen in Figures 5.9 and 5.10.
First, note that the process execution in these picture is stopped after the visit of the second loop. The first picture shows the visiting graph used in the example, as well as the propagation tree. This propagation tree is slightly different

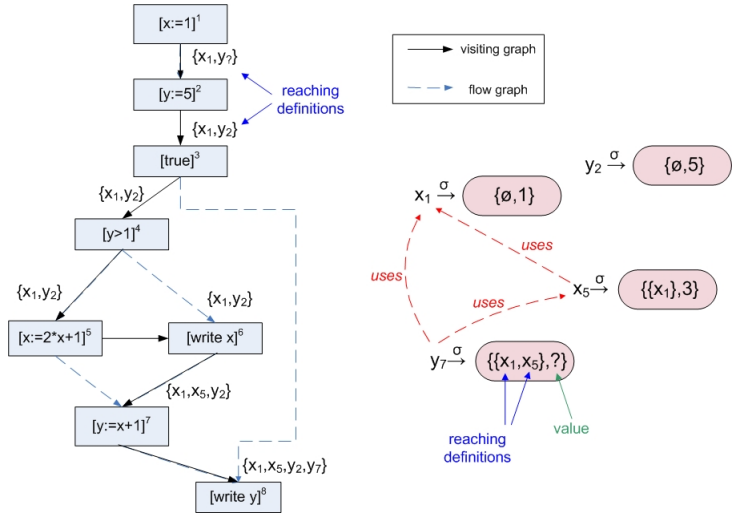


Figure 5.8: Use of the *sigma* function for Constant Propagation

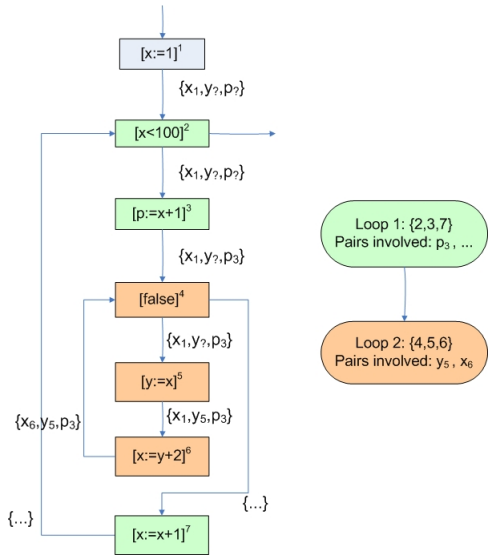


Figure 5.9: Visiting graph and propagation tree

from the ones seen with the analyses from the bit-vector framework: here the nodes keep track of the definition pairs involved in the loop, so the propagation does not reach unwanted node outside the loop involved.

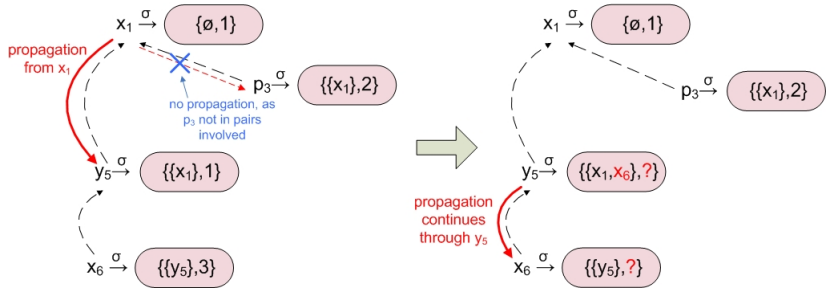


Figure 5.10: Propagation mechanism for Constant Propagation

The second picture shows the propagation of x_6, starting from x_1 to each of the pairs linked to $\sigma(x_1)$ (and in the set of pairs involved): the set of reaching definitions is updated and the value set to "?". Then the propagation continues through y_5, because its value has turned into "?": thus the linked variables' value must also be set to "?" and propagated.

Hence, this mechanism gives the same information as the Constant Propagation Analysis, by using the propagation algorithm for Reaching Definitions Analysis computed earlier, and adding a constant propagation mechanism using constraints.

5.3.4 Detection of Signs approximation

The Detection of Signs Analysis is the last analysis to be approximated. This analysis is similar to the Constant Propagation Analysis, except that instead of mapping a pair ($\mathbf{Var} \times \mathbf{Lab}$) to a set of reaching definitions and a value, the evaluation function maps it to a set of reaching definitions and a set of signs.

Thus the Detection of Signs Analysis uses the function:

$$\sigma_{ES} : (\mathbf{Var} \times \mathbf{Lab}) \rightarrow (\mathcal{P}(\mathbf{Var} \times \mathbf{Lab}) \times \mathcal{P}(\{\text{-}, 0, +\}))$$

The use of the σ_{ES} function, the propagation mechanism and the use of propagation tree nodes, are built in the same way as with the Constant Propagation Analysis (see Section 5.3.3).

The single (but important) change is that instead of propagating "?" when the values are different than in Constant Propagation, the Detection of Signs

Analysis needs to propagate the union of the different set of signs. So the propagation mechanism used here only propagates the Reaching Definitions to the variables linked in the current constraints, and the values are re-evaluated from this new set of Reaching Definitions, instead of being only updated to "?". Then, of course, the propagation continues to the following linked variables.

5.3.5 Type of analysis and accuracy of the results

As has been pointed out during the description of the propagation algorithm and the definition of the parameters for the different analyses, the main issue of this algorithm is that, to propagate the entry information coming from a loop's body, there must be a way to figure out until where the data must be propagated through the loop's body. Concretely, using the terms of Section 5.3.2.1, there must be a way to know until where the *entryDown* information in the loop's blocks to add (or remove) the *entryUp* information.

An interesting point is the fact that analyses from the bit-vector framework defined previously give very accurate data with the propagation algorithm. In fact, using this algorithm, these adapted versions of the analyses will give *exact* results, due to the fact that the entry/exit informations are proper to each block and not used for any choice on the transition function to apply. An experimental study on all the benchmark programs used previously in the benchmark suite has been made to confirm that running the propagation algorithm for these analyses gives exactly the same results as using a classic algorithm such as the MFP algorithm (the results are located in Appendix B).

The two other analyses considered, Constant Propagation and Detection of Signs, have been computed with a modified algorithm using constraints in order to keep this accuracy of the results. Again, experiments on the different test programs have confirmed that the propagation mechanism also gives *exact* results in these two cases, i.e that these two adapted versions of the analyses returns the same informations (though arranged differently) as the original Constant Propagation and Detection of Signs using the MFP algorithm.

On the other hand, in the case of the Strongly Live Variables Analysis (Section 3.3.7), it is not possible to apply this algorithm. Indeed, this analysis uses the entry information $SLV_{exit}(l) = analysis_{\circ}$ to make a choice on whether or not using the *kill* and *gen* sets. In the case this choice is made in one way at the first computation (e.g. the variable is thought to be faint and it is thus not added to the set of live variables), if the algorithm changes the entry information $SLV_{exit}(l)$ during the propagation, forcing the other choice, then the exit information would be completely changed, and all the following blocks' entry/exit

information would need to be re-computed.

Hence, it can be conclude that there must be a pattern describing the different analyses that can use this kind of algorithm, either the direct and simple algorithm used with the analyses from the bit-vector framework, or the modified algorithm using constraints used for the last two analyses. Thus, an interesting future work could be to investigate and try to describe this class of analyses, and perhaps find a way to derive some constraints from the different equations characterizing the analyses (see Chapter 9).

5.3.6 Comparison for data flow analysis

As seen in the previous section, for some analyses, the propagation algorithm gives exactly the same results as the classical algorithms. An interesting point is then to see the running time of the different algorithms, in order to compare the different algorithms and also to see if the propagation algorithm can be fast enough to fit the metric-based approach requirements.

The comparison is performed on the same benchmark programs as in the evaluation of the benchmark suite at Section 4.3.3.5. The Copy Analysis has been chosen as an example, using two classical algorithms, the MFP algorithm and the abstract worklist algorithm using iteration through strong components, defined in [16], and the propagation algorithm from Section 5.3.2.1. The results are shown under different views in Figure 5.11 and Figure 5.12. The full table can be seen in Appendix A.

The first remark that can be made about this figure is that for the smallest programs, the running time of the algorithms are so small that the comparison does not appear to be relevant, though in that kind of case none of the algorithms appears to be far behind any other.

The most interesting point is that, though the two classical algorithms are running in the same scale of time (with better results for the abstract worklist algorithm), the propagation algorithm gives most of the time satisfying results. The figures for the two biggest benchmark programs (2 and 10) clearly show that the propagation algorithm is faster than the two other algorithms.

These results are very promising, as they show that the propagation algorithm can be used in the metrics' calculation without significantly increasing the total optimizing time.

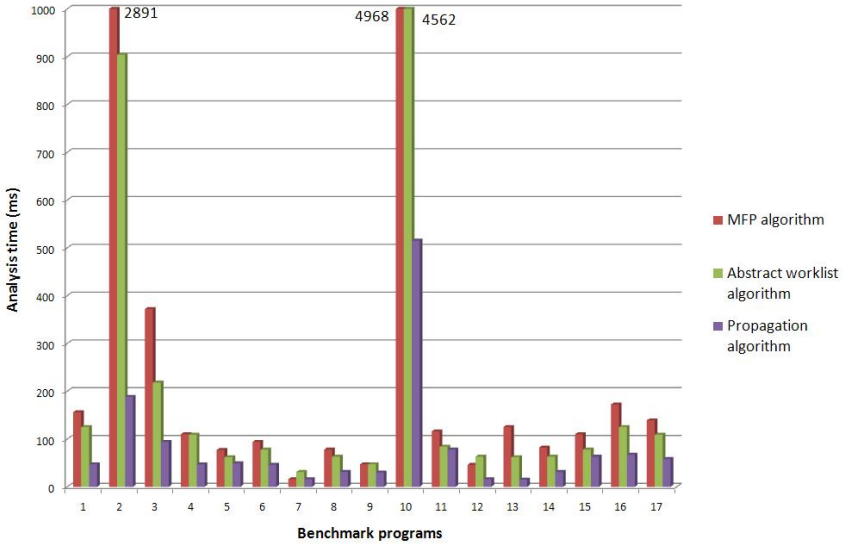


Figure 5.11: Comparison of the Copy Analysis computation using different algorithms

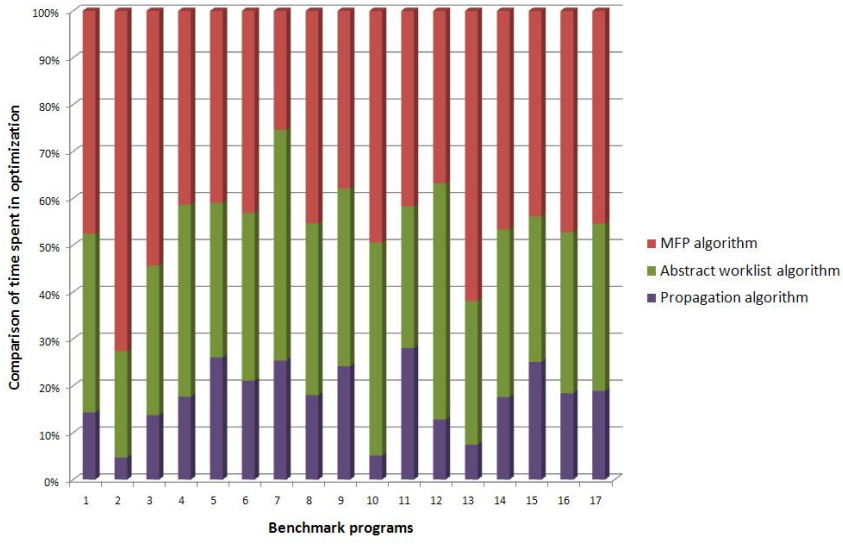


Figure 5.12: Relative comparison of Copy Analysis using different algorithms

5.4 Definitions of the metrics

Once the different analyses have been approximated, the results can be used to calculate the metrics that will be used in the phase-ordering algorithm (see Section 5.5). These metrics are all implemented in the Phase Manager. Depending of the type of analysis, these metrics can give very accurate results on where the transformation will be efficient, but they are *all* designed to be over-approximations, i.e. a metric should not have a value of zero if the transformation can be efficient somewhere.

5.4.1 Metric for Copy Propagation

The metric CP represents the Copy Propagation transformation. Copy Propagation deals with variables that have the same values as other variables. This is due to the occurrence of copy assignments (assignments of type $[x := y]^l$). As shown in Section 3.4.3, the Copy Propagation transformation replaces these variables by their copy variables, thus making copy assignments become dead assignments. The main improvement of the program is that target variables become dead, therfore copy assignments are removed by the transformation.

The metric for the Copy Propagation transformation represents the number of copy assignments that could be removed using the transformation. This metric uses the result from the Copy Analysis performed using the propagation algorithm. It starts also by computing the ud- and du-chains, using the propagation algorithm on the Reaching Definitions Analysis (see Section 5.3.2.3). Then it follows closely the different steps involved in the transformation itself. For each of the copy assignments $[\texttt{x:=y}]^l$ of the program, it:

1. Gets the set of blocks using the copy assignment from the Definition-Use chains.
2. If the copy assignment is still valid at each of these blocks, increases the metric's value by 1.

Indeed, in this case, the transformation would delete the copy assignment (and replace the variables used by the copy variable). To know if the assignment is still valid, the entry information $approx_\circ$ from the Copy Analysis is used.

As shown in Section 5.3.5, the result of the propagation algorithm for the Copy Analysis and the Reaching Definitions are exact, so this metric gives the *ex-*

act number of copy assignments that will be removed by a Copy Propagation transformation.

5.4.2 Metric for Common Subexpressions Elimination

The metric CSE represents the Common SubExpressions Elimination transformation. This transformation is only concerned with expressions, and their occurrences in the program. Indeed, the more often an expression is computed in the program, the more often the Common SubExpression Elimination transformation could be applied to reduce the number of computation by introducing a temporary variable.

The metric for the Common Subexpressions Elimination represents the number of available expressions re-used in the program. It uses the propagation algorithm with the parameters for the Available Expression Analysis to compute the set of available expressions at each block's entry. Then, for each block of the program, it:

1. Gets the set of non-trivial expressions involved in the block.
2. If an expression is already available, increases the metric's value by 1.

On all the available expressions re-used in the program, some will have to introduce new temporary variables, e.g. when the expression is part of a loop, and thus will not be as interesting as those where only a replacement is made, but this metric gives a good over-approximation of the places where the transformation may be beneficial.

5.4.3 Metric for Code Motion

The metric CM represents the Code Motion transformation. The transformation tracks *loop invariants* and moves them out of the loop in order to reduce the number of computations of these invariants. Thus, most of the time, the transformation improves the program for each of the loop invariants.

The metric for the Code Motion transformation represents the number of *loop invariants* in the whole program. It uses the result from the propagation algorithm used with the parameters from the Reaching Definition Analysis, as well as the new version of Use-Definition and Definition-Use chains (see Section 5.3.2.3).

For all assignment $[\texttt{x:=y}]^l$ in loops, it:

1. Using the UD-chain, checks whether all variables used are defined outside the loop, or if not checks whether only one definition reaches the variable and that definition is already a loop-invariant.
2. As the execution can only leave a while loop by rendering the boolean condition false, the block containing $[\texttt{x:=y}]^l$ always dominates all loop exits, so there is nothing to check for this point.
3. Then, first checks if there is no other assignment to x in the loop. This is done by looking at the entry information $approx_\circ$ and checking if it does not contain other results for x among the labels in the loop.
4. Finally, using the DU-chain, check if no use of x in the loop is reached by any definition of x other than $[\texttt{x:=y}]^l$
5. If every check is OK, then mark the assignment as a *loop invariant*, and increase the metric by 1.

This metric hence gives the *exact* number of loop invariants of the program where the Code Motion transformation could be applied.

5.4.4 Metric for Constant Folding

The Constant Folding transformation uses results either from the Reaching Definition Analysis or the Constant Propagation Analysis. The metric for Constant Folding has been based on Constant Propagation, since it has been explained in Section 3.4.1 that this analysis yields better results.

The metric CF represents the number of variables that could be replaced by a constant value. This metric uses the results from the propagation algorithm for Constant Propagation. It is calculated by iterating through the blocks of the program and, for each of them, it:

1. Gets the set of variables used in the block
2. For each of these variables gets the assignments (of the same variable) reaching the block, using Reaching Definitions Analysis results
3. Gets the value of each of these variable definitions from the σ map (from Constant Propagation Analysis Section 5.3.3), and gets the value of the

variable used. If this value is constant (i.e different from "?"), increases the metric's value by 1.

As explained in Section 5.3.5, the σ mapping and the results from the propagation analysis used with the Reaching Definitions Analysis parameters gives exact data, so it can be concluded that this metric returns *exactly* the number of variables that have a constant value in the program. Once again, this claim has been experimentally verified using the Phase Manager implementation and the different test programs.

5.4.5 Metric for Elimination with Signs

The metric ES represents the Elimination with Signs transformation. This transformation aims at statically computing expressions when the signs of the operands are known. It uses the Detection of Signs Analysis.

The metric for Elimination with Signs uses the σ_{ES} map from the propagation algorithm for Detection of Signs (see Section 5.3.4) to determine how many expressions could be statically computed. For each expression of the program, it:

1. Looks for non-trivial boolean expressions
2. Gets the sets of signs of all the operands of the expression, using the σ_{ES} map.
3. If these sets of signs match, the result of the expression's computation can be guessed, so increases the metric's value by 1.

As for the metric for Constant Propagation Analysis, the mechanism used to calculate this metric gives exact results. Thus, the metric will show the number of places where the Elimination with Signs transformation will be able to statically compute the expressions.

5.4.6 Metric for Dead Code Elimination

The last metric to be computed is the metric DCE. This metric represents the Dead Code Elimination transformation. This transformation simply aims at deleting the variables declared *dead* in the program, and can be extended to

delete the variables declared *faint* as well (see Section 3.4.4).

The metric for Dead Code Elimination has been designed to represent the total number of assignments that would be removed by the Dead Code Elimination transformation. A simple way to calculate this metric would be to use the result of the propagation algorithm with the parameters for the Live Variable Analysis, which gives exact results. However, the Dead Code Elimination transformation is applied using the Strongly Live Variables Analysis in the compiler, because this analysis is available in the compiler and gives better results.

Hence, in order to count the assignments of faint variables as well, a mechanism to find faint variables must be added. The metric is calculated by iterating through the blocks of the program. Then, for each assignments $[x := a_1]^l$, it:

1. Gets the sets of live variables at the exit of this block.

2. If this set does not contains x_l, declares the variable dead, and increases the metric's value by 1.

3. Look for any variable becoming faint because of x_l being dead. This uses the auxiliary algorithm defined below.

The auxiliary algorithm aims at finding the faint variables from a dead/faint assignment $[x := a_1]^l$. For each of the variables y in $FV(a_1)$:

1. Get the set of assignments defining y, using the UD-chain.

2. For each of these definition $[y := a_2]^{l'}$, get all the places where y is used, from the DU-chain. If all these places are assignments of already declared dead or faint variables, declare the variable faint and increase the metric's value by 1. Then look for any variable becoming faint because of $y_{l'}$ being faint.

Using these UD- and DU-chains, this metric gives the *exact* number of dead *and* faint assignments, though finding faint variables takes a little more time than just using the results from the propagation algorithm with the parameters from the Live Variables Analysis.

5.5 Metric-based phase-ordering algorithm

The different metrics defined previously aim at creating a dynamic order of transformations during the optimization process in the compiler. As the Phase Manager is responsible for the ordering of the different phases, it will compute the metrics and analyze them to choose in which order the different transformations should be called.

The following sections deal with the design of an algorithm that will be used to determine which transformations should be called and in which order.

5.5.1 Algorithm used

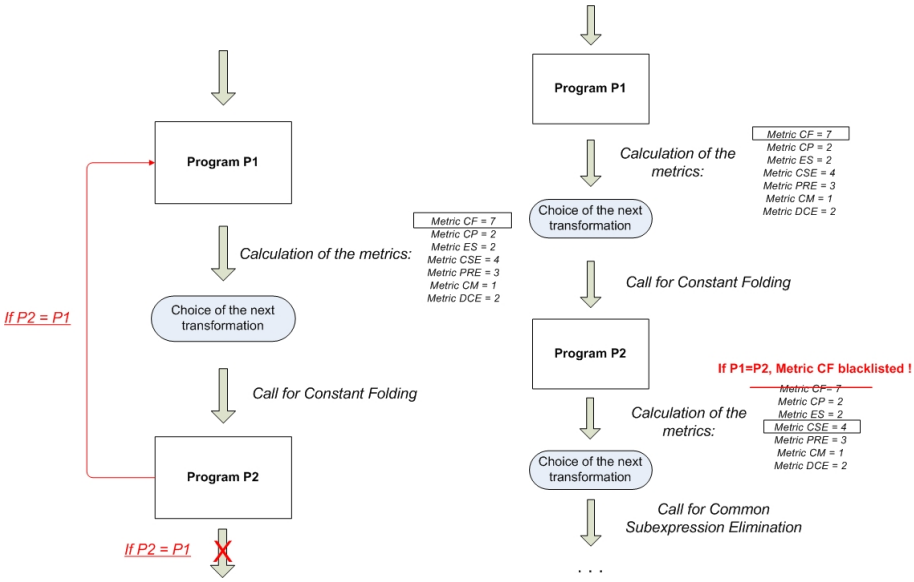
In order to define the algorithm to compute an efficient order of optimizations, two points must be determined:

1. given the values of the different metrics, which transformations should be called first, and
2. when should the manager stop the execution of the algorithm.

The *first* issue can seem easy to solve, but in fact it raises some problems. Indeed, the obvious way to compute the next transformations to be called is to look at the different values of the metrics and to take the transformation(s) associated with one of the metrics with the highest value, as all the metrics' values are on the same scale.
But something else must be taken into account. Because of the fact that metrics are defined by over-approximation (see Section 5.4), it may happen that a metric has a non-zero value that corresponds to blocks of interest in the program, but that in fact the corresponding transformation has been judged applicable on these blocks while it is not. Then the value of this metric corresponding to these blocks never decreases if the transformation is applied, and this situation can result in a deadlock, as can be seen in Figure 5.13(a).

That is why the algorithm should include a mechanism for remembering the metrics corresponding to transformations that are not efficient on the program: hence, whenever some transformations do not change the program, their corresponding metric is added to a *blacklist*. This *blacklist* is updated at every phase, by removing all the metrics that have changed since the preceding phase:

indeed, if a blacklisted metric has changed, it means that there may be other optimizations to perform on the program that were not applicable before, and so the transformation may be beneficial again. In this case the deadlock is avoided, as illustrated in Figure 5.13(b). This mechanism is also a good way to allow the manager to skip the transformations that have proved to be inefficient, while their metric still has a non-zero value, due to over-approximation.



(a) Case of deadlock (b) Deadlock avoided

Figure 5.13: Use of the blacklist

The *second* point is also relatively important: the manager needs a way to decide whether or not it should continue to call optimizations on the program. This condition is closely related to the different issues that appeared when analyzing the *first* point. Indeed, as the transformations are ranked by the values of their metric, the condition of termination is to have all the metrics set to zero. But as described earlier, some metrics may have encountered cases where they could not decrease because of over-approximation in their definition. That leads to the algorithm termination being based on all the metrics either set to zero or blacklisted.

However, as the degree of optimization of the program at the end of the process is the primary concern, another issue has to be accounted for. As explained before, some of the metrics may have been blacklisted to avoid useless transfor-

mation calls (and deadlock), and are supposed to get out of *blacklist* once their value is changing. But there can be some cases where the value of the metric is not changing while the metric should be un-blacklisted.
Consider two metrics m_1 and m_2, respectively with values 2 and 1. The manager will then decide to call the transformation corresponding to m_1. In the case that metric m_1 was an over-approximation and the transformation does not change the program, the metric m_1 will be blacklisted and the transformation corresponding to m_2 will be applied. Assume that the metrics' values are now $m_1 = 2$ and $m_2 = 0$. The metric m_1 will not be unblacklisted, but there is a chance that the transformation corresponding to m_2 removed the over-approximation on m_1 and that now the transformation corresponding to m_1 will have some effects on the program. Therefore, the algorithm must include a way to give a "*second chance*" to these transformations in this case.
The mechanism used to solve this problem checks whether the *blacklist* is empty at the end of the execution. If the algorithm has reached the end of the execution with an empty *blacklist*, it means that all the metrics have a zero value, and then the issue raised before is avoided.
Otherwise, the "*second chance*" must be given, and the *blacklist* should be emptied. So the mechanism includes the creation of two others sets, called *notToBeDeBlacklisted* and *toBeDeBlackListedIfChange*. The set *notToBeDeBlacklisted* represents the metrics that have been blacklisted since the last change on the program: these metrics should in fact not be removed from the *blacklist* until the program is changed again, as it is known that they are inefficient on the actual version of the program.

At the end of the "*first chance*", the last set *toBeDeBlackListedIfChange* will contain the same metrics as the set *notToBeDeBlacklisted*, in order to remember which metrics are supposed to be directly removed from the *blacklist* whenever the program changes.

The actual algorithm can be seen in Figure 2. It uses several parameters:

- three instances of the program (including the algorithm input) **prg**, **prg2** and **prgAtEnd**.
- two mappings between the metrics and their values **metricValues** and **metricValuesOld**.
- three sets of metrics **blacklist**, **notToBeDeBlacklisted** and **toBeDeBlackListedIfChange**.
- a metric name **op** used to identify the associated transformation(s) to be applied.

```
INITIALIZATION STEP:
Initialize blacklist and metrics' maps to ∅;
prgAtEnd = clone(prg);
op = updateMetricsMap(metricValues,metricValuesOld,blackList);

ITERATION STEP:
outerloop: while true do
    while op ≠ −1 do
        prg2 = clone(prg);
        callOptimizations(op,prg);
        if prg = prg2 then
            blacklist = insert(op);
            notToBeDeBlacklisted = insert(op);
            op = findHighestMetric(metricValues,blackList);
        else
            notToBeDeBlacklisted = ∅;
            blacklist = blackList ⋂ toBeDeBlackListedIfChange;
            toBeDeBlackListedIfChange = ∅;
            metricValuesOld = insert(metricValues);
            op =
            updateMetricsMap(metricValues,metricValuesOld,blackList);
        end
    end
    if blacklist ≠ ∅ then
        if prg = prgAtEnd then
            break outerloop;
        else
            blackList = blackList ⋂ notToBeDeBlacklisted;
            toBeDeBlackListedIfChange = toBeDeBlackListedIfChange ⋃
            notToBeDeBlacklisted;
            notToBeDeBlacklisted = ∅;
            prgAtEnd = clone(prg);
            op = findHighestMetric(metricValues,blackList);
        end
    else
        break outerloop;
    end
end
```

Algorithm 2: Metric-based phase-ordering algorithm

Additional functions are used within the algorithm as well:

- the function *updateMetricsMap()* computes and updates the mapping between the metrics and their current values, and returns the identifier of the metric with the highest value. It also updates the blacklist by making the difference between the two mappings of metrics' values (old and new) and returning the blacklist where all the metrics whose value has changed have been removed
- the function *findHighestMetric(...)*, given a mapping with metrics' values and a *blacklist*, returns the metric's name whose associated transformation(s) will be used on the program. As explained before, the function returns the name of the non-blacklisted metric with the highest value. If no non-blacklisted metric is greater than zero then the function returns -1.
- the function *clone(...)* returns an exact clone of the specified instance of the program.
- the function *callOptimization(...)* launches the transformation(s) given by the specified metric on the specified instance of the program. In the implementation, this function takes another parameter that allows a control on whether the transformation(s) called should uses the analysis computed in the metric's calculation or just recompute the analysis with the classical algorithms.
- the function *insert(...)* that inserts one or several elements specified as parameters into a set or a mapping.

5.5.2 Termination of the algorithm

As two *while* loops are involved in the algorithm, it is interesting to see whether this algorithm can always terminate, as otherwise it becomes useless. This part contains a small proof that the algorithm will eventually terminate, with the following assumptions concerning the degree of transformation of the program d:

(i) it is assumed that there exists a maximum value for d where the program is as transformed as possible, and it is not possible to optimize it further, i.e.,

$$\exists d_{max} : \forall d, d \leq d_{max}$$

(ii) it is assumed that every metric m involved in this algorithm is associated with transformations t_m that either improve the degree of transformation of the program or let it unchanged, i.e.,

$$\forall m, t_m \text{ applied} \Rightarrow d_{after\ t_m} \geq d_{before\ t_m}$$

These two assumptions are valid if it is not possible to have several transformations T_1,...,T_n such as the program generated after the sequence $T_1 \cdot ... \cdot T_n$ is equal to the original program. It is true for the optimizations considered in this thesis, simply because they are shown to reduce the number of instructions executed by the program (or at least not increase it) whenever they are applied (see Section 7.2), and this number of instructions has an obvious lower bound of 0. However, the general concept is addressed in Chapter 9.

To prove that the algorithm eventually terminates, one must prove that:

(a) the execution gets out of the inner loop, i.e., eventually reaches the inner loop condition in a state where $op = -1$. For this matter, the last statement of the loop's body (call to *findHighestMetric(...)*) must assign the value -1 to **op**.

(b) the execution gets out of the outer loop, i.e., eventually reaches one of the two statements **break** *outerloop*.

To solve the first point, assumption (ii) can be used. Two cases are then possible: either the degree d does increase, or the degree d is unchanged, and then, according to the algorithm, the metric m is placed into the *blacklist*. The execution can then reach three different situations:

1. if every metric m has a value of zero, then the blacklist is empty and the last function call *findHighestMetric(...)* returns -1
2. if no non-blacklisted transformation t_m makes d increase anymore, then each metric m has either a value of zero or is in the blacklist, thus the last function call *findHighestMetric(...)* returns -1, too.
3. if d keeps increasing, following assumption (i), the execution eventually reaches a point where $d = d_{max}$ where no more transformation t_m can improve the program: thus it is either like case 1 or 2.

Consequently, in the three cases, the value of **op** will eventually be -1, and the execution will go out of the inner loop.

The second point deals with the outer loop. Once the execution gets out of the inner loop, if the blacklist is empty or the program did not change since the last time it got there, one of the **break** *outerloop* is directly reached. Otherwise, either

- d_{max} is reached, and if the *blacklist* is not empty yet, the execution goes back to the inner loop in the case 1 or 2. Then, once out of the inner loop again, it directly quits, and, as the program did not change ($prgAtEnd = prg$), the first **break** *outerloop* is reached.
- d_{max} is not reached yet and the *blacklist* is not empty: the algorithm gives a *"second chance"* to the blacklisted metrics and it enters the inner loop again. The program then keeps being optimized, until d_{max} is reached or no transformation from non-zero metrics can optimize the program better. As the metrics are defined by over-approximation, the second case should not happened without d_{max} being reached, because then the metrics that could optimize the program to d_{max} should not have a value of zero.

Chapter 6

Evaluation of the metric-based phase-ordering

This chapter deals with the evaluation of the metric-based approach of the phase-ordering problem. The first section describes the comparison made between the metric-based approach and the best regular expressions from the benchmark suite. The second section introduces an analysis of the dependencies between the transformations, and how the phase-ordering algorithm can be modified according to it. This chapter aims at giving the reader an overview of the performance of the metric-based approach defined in the previous chapter, as well as an idea of a potential improvement of this approach.

6.1 Evaluation: Comparison with results from benchmark suite

This section deals with the comparison between the metric-based phase-ordering and the results from the benchmark suite using regular expressions. The tests have been run on the same benchmark programs as during the evaluation of the benchmark suite. The characteristics of the machine used for these tests are not of primary importance, as the main interest of this evaluation is the comparison

of the two approaches.

In the metric-based approach, a propagation algorithm is used to approximate the analysis during the calculation of the metrics. On several cases, these analyses give exact results, which can then be used when performing the transformations on the program. Thus, the next two subsections are organized as follows: the first subsection is about the metric-based approach without reusing the results from the metrics' computation, while the second subsection deals with the same approach with a reuse of the analysis results already computed.

6.1.1 Without using analysis results from metrics' computation

In this section, the results of the metric-based phase-ordering will be presented and compared to the results from the benchmark suite from Section 4.3.3.5. While the benchmark suite uses a regular expression generator which includes probabilities in generating the different expressions, the metric-based compilation will always compute the same order of transformations for the same test program, as the algorithm is totally deterministic.

The results show that the phase-ordering using metrics optimizes the program as much as the best regular expressions would do. The tables of data can be found in Appendix A.3. As it can be seen in Figure 6.1, the overall time spent is most often better for the metric-based approach.

As the values from Bench Program 10 are much higher than the others, the highest part has been cut in the graph. The lower part of the columns for the metric-based approach represents the time spent in the metrics' calculation. In some case the time spent on the metric's calculation is a significant percentage of the overall time, but this increase of time is counterbalanced by the fact that the optimization is using less transformations than with the regular expression-based approach. Indeed, only one regular expression is giving better results in terms of number of transformations used (for Bench Program 4). This comes from the over-approximation in one of the metrics, namely the metric for Common Subexpression Elimination.
The only case where the best regular expression is performing better is for the Bench Program 10, where the metric-based algorithm takes 36% more time than the best regular expression. In this case, the best regular expression compiles the optimal program using as many transformations as the metric-based approach; the overhead in metric-based compilation is therefore coming from the metrics' calculation.

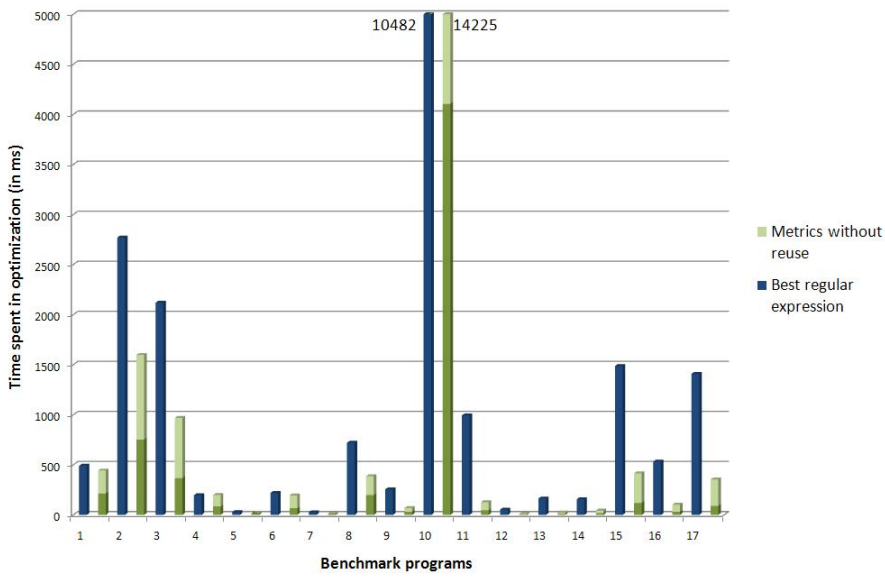


Figure 6.1: Comparison of the overall time spent in optimizing

6.1.2 Using analysis results from metrics' computation

During the calculation of the different metrics, some analyses are approximated, as explained in Section 5.3, using a propagation algorithm. In particular, in some cases, the use of the propagation algorithm gives exactly the same results as the classic algorithms used when the transformations are actually applied. Hence, in these cases, the analysis' results can be re-used instead of computing the analysis solutions again with these algorithms, first because they are already available, and also because, in case a re-computation is needed during the transformation itself, the propagation algorithm has been shown to be faster (see Section 5.3.6).

Thus, the phase-ordering mechanism based on metrics has been applied using the analysis' results from the metrics' computation, and it has been compared to the one that does not reuse these results. The table of results of this comparison is shown in Appendix A.3. This comparison does not take into account the number of instructions executed nor the number of transformations, as they are obviously the same, since nothing changes in the optimization process itself. Figure 6.2 shows a complete comparison, including the results from the benchmark suite, and the two cases of metric-based algorithm.

The time spent on the metrics' calculation for both cases is approximately the

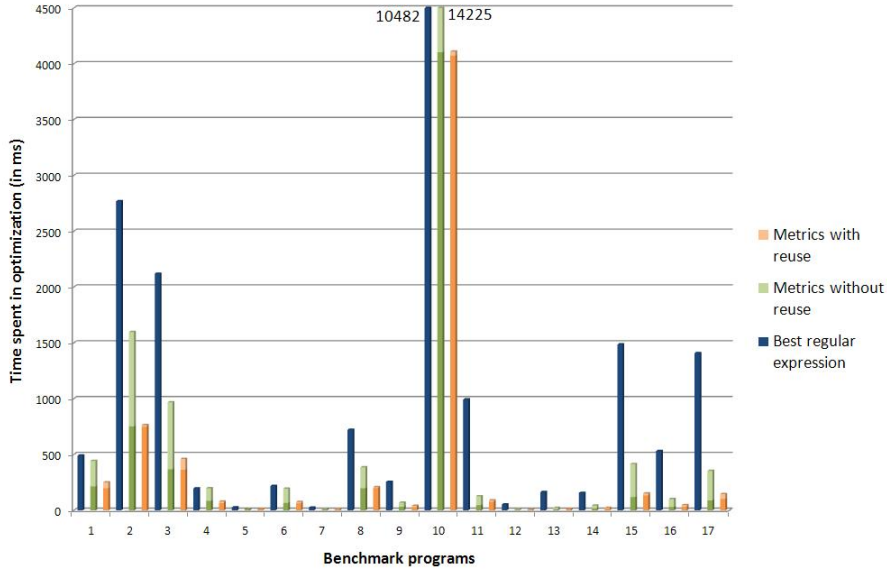


Figure 6.2: Complete comparison of the overall time spent in optimizing

same, as could be expected. Indeed, though the different analysis' results coming from the metrics' computation are reused, the computation process of the metrics themselves is not changing at all. Concerning the overall time spent in the optimization, the algorithm using metrics and reusing the analysis results is performing better than when the analysis solutions are re-calculated in the transformations using classical algorithms, confirming the good performance of the propagation algorithm. The case of Benchmark Program 10 is again cut on this graph; however, the metric-based phase-ordering with reuse of the metrics' results is only taking 39% of the time the best regular expression needed.

This graph concludes the comparison between the two considered approaches: the metric-based approach is clearly giving better result than the regular expression approach. More precisely, the best regular expression gives often worse results than the metric-based phase-ordering. The number of transformations used in the metric-based approach is close to the optimal one, though the fact that the metrics are defined by over-approximation may still introduce some unnecessary transformations.

6.2 Dependencies between transformations

As it can be seen in the previous section, the time spent on the metric calculation is an important parameter in the speed of the optimization process. In order to reduce the metrics' calculation time, the approach considered in this section is to study the different connections between the transformations, and then try to derive from these dependencies a mechanism in which some metrics would not be re-computed after specific transformations, in the case where the value is known not to change.

As explained in Section 2, the interactions between the different transformations is one of the most interesting topics that can be considered when dealing with phase-ordering in compilers. It is here possible to draw an overview of the interactions between the transformations involved because they are all well described and understood.

A last point concerns the Precalculation Process. Indeed, in the current phase-ordering mechanism, the Precalculation Process is applied after each transformation, because it is a very cheap transformation, almost as cheap as the metric that could be designed for it. However, in order to perform this non-recomputation of the metrics, Precalculation must be treated like all the other transformations. Indeed, it interacts with every transformations and provokes changes in all the other metrics. As a consequence there would be no case where a metric is sure not to have changed if Precalculation is always called after every transformation.

This section first describes the metric for the Precalculation Process, then establishes which transformation can enable or disable other transformations, and finally describes the experimental study made in order to confirm the interactions found and their impact on the metrics, in order to avoid useless re-computation of metrics. At the end of the section, a performance comparison is made between this new way to compute the metrics and the previous version.

6.2.1 Metric for Precalculation Process

The metric PRE represents the Precalculation process. This transformation aims at statically compute constant expressions, remove skip statements and simplify both if and while statements whenever the condition is constant. It uses no analysis, and thus can be perform in a linear time, as the program to transform is skimmed through only once.

As can be seen in Section 3.4.7, this process mainly deals with four parameters:

- the skip statements
- the non-trivial expressions where only constant values are involved
- the if statements with a constant condition
- the while loops with a *false* condition

⇒ **Definition of the metric:** The metric PRE, used to rank the need of using the Precalculation process, is computed by counting the number of skip statements, of non-trivial expressions where only constant values are involved, of if statements with a constant condition and of while loops with a *false* condition.

No approximation are involved in this metric, so it obviously finds the exact number of places the transformation could be applied.

6.2.2 Overview of the dependencies between transformations

In some cases, a transformation can perform some changes on the intermediate representation such that the others transformations' efficiency is modified, i.e the different places where this transformations can be applied are not the same after this transformation. In these cases, these other transformations will be able to perform either better or worse than they would have done if the first transformation would not have been called. In other cases, performing the first transformation cannot change the results of some specific transformations, whatever the program given as input is.

In [24, 25], Whitfield and Soffa used their theoretical model to conclude on the dependencies of some transformations. In the remaining of this section, a more practical approach is used to define these interactions, in an attempt to draw a connection table. Then this connection will be compared to Whitfield and Soffa's conclusions on the transformations in common.

For the sake of readability, only the interactions of Constant Folding with the other transformations are described in this section. The remaining transformations can be found in Appendix C.1. Then, the connection table is summarized,

"tested" experimentally and used in the last subsections in an attempt to improve the update of the different metrics.

6.2.2.1 Interactions of Constant Folding

This part deals with the interactions of Constant Folding. It describes the different effects that the transformation can have on all the other ones, insisting on when the Constant Folding transformation is not modifying any results for some transformations. The (+) symbol means that there is a connection from Constant Folding, where (-) means there is not.

- **Dead Code Elimination (+).** Constant Folding is obviously connected to the Dead Code Elimination transformation: indeed, whenever Constant Folding is applied, it replaces some variables x by their constant value, potentially making assignments to x dead, as can be seen in the following example:

  $$[\texttt{a:=5}]^1; [\texttt{b:=a*a+3}]^2; \texttt{write } [\texttt{b}]^3$$

  In the above program, Dead Code Elimination cannot be applied, but applying Constant Folding enables it, as illustrated here:

  $$\begin{gathered} \Rightarrow_{CF} [\texttt{a:=5}]^1; [\texttt{b:=5*5+3}]^2; \texttt{write } [28]^3 \\ \Rightarrow_{DCE} \texttt{write } [28]^3 \end{gathered}$$

- **Common Subexpressions Elimination (+).** Constant Folding also enables Common Subexpressions Eliminations: as an example, the latter is not capable of recognizing the expressions `a+b` and `a+3` to be the same, even when `b` is constant and has a value of 3. Thus, after a Constant Folding, the two expressions will be `a+3`, and the Common Subexpressions Elimination will be applicable.

- **Precalculation Process (+).** Constant Folding obviously enables the Precalculation Process. For example in the program:

  $$[\texttt{a:=5}]^1; [\texttt{b:=a*a+3}]^2$$

  the Precalculation Process is useless, while after Constant Folding it can be applied:

  $$\begin{gathered} \Rightarrow_{CF} [\texttt{a:=5}]^1; [\texttt{b:=5*5+3}]^2 \\ \Rightarrow_{PRE} [\texttt{a:=5}]^1; [\texttt{b:=28}]^2 \end{gathered}$$

Constant Folding can also enables Precalculation by replacing a boolean variable in a condition (in if or while statements) by their boolean constant value, allowing Precalculation to simplify the statement.

- **Code Motion (+).** The Constant Folding transformation is also interacting with the Code Motion transformation. In the program:

$$\texttt{while } [\texttt{x<3}]^1 \texttt{ do } [\texttt{x:=1}]^2; [\texttt{p:=x}]^3 \texttt{ od}$$

Code Motion cannot be applied at label 3, because it uses x assigned in the loop, and which is not an invariant (since x is used in the condition as well). However, after Constant Folding, the label 3 can be moved out of the loop:

$$\begin{array}{c} \Rightarrow_{CF} \texttt{while } [\texttt{x<3}]^1 \texttt{ do } [\texttt{x:=1}]^2; [\texttt{p:=1}]^3 \texttt{ od} \\ \Rightarrow_{CM} \texttt{if } [\texttt{x<3}]^4 \texttt{ then } ([\texttt{p:=1}]^5; \texttt{while } [\texttt{x<3}]^1 \texttt{ do } [\texttt{x:=1}]^2 \texttt{ od}) \texttt{ fi} \end{array}$$

- **Copy Propagation (+).** Copy Propagation is influenced by the use of Constant Folding. Indeed, Copy Propagation deals with the replacement of copy variables and delete the copy assignments if possible. If the variables used in the copy assignments are replaced by their constant values, Copy Propagation cannot be applied while it could before. For example, Copy Propagation can be applied in the following program:

$$\begin{array}{c} [\texttt{x:=1}]^1; [\texttt{p:=x}]^2; \texttt{write } [\texttt{p}]^3 \\ \Rightarrow_{CP} [\texttt{x:=1}]^1; \texttt{write } [\texttt{x}]^3 \end{array}$$

while after Constant Folding it cannot be applied anymore:

$$\begin{array}{c} [\texttt{x:=1}]^1; [\texttt{p:=x}]^2; [\texttt{write p}]^3 \\ \Rightarrow_{CF} [\texttt{x:=1}]^1; [\texttt{p:=1}]^2; \texttt{write } [\texttt{1}]^3 \\ \Rightarrow_{CP} [\texttt{x:=1}]^1; [\texttt{p:=1}]^2; \texttt{write } [\texttt{1}]^3 \end{array}$$

- **Elimination with Signs (-).** The only transformation not evolving after Constant Folding is the Elimination with Signs transformation. This comes from the fact that, if a variable x is replaced by its constant value n, the Detection of Signs Analysis will find that $\sigma(x, l) = sign(n)$, while without Constant Folding it will have to propagate the sign of n to x. So the use of Constant Folding makes the analysis easier, but the results will be the same. The Elimination with Signs transformation will not get more information from the Detection of Signs Analysis with or without the use of Constant Folding.

6.2.2.2 Summary of the dependencies

The following table (Figure 6.3) sums up the different dependencies between the transformations used. Of course each transformation is potentially influenced by itself, as using a transformation can decrease the efficiency of a future re-use of this transformation. The interesting part is of course the transformations that *does not* influence some other transformations, so metrics do not have to be re-computed.

	CF	CSE	CP	DCE	CM	ES	PRE
CF	√	√	√	√	√		√
CSE	√	√	√	√	√		√
CP	√	√	√	√	√		
DCE	√	√	√	√	√	√	√
CM	√	√			√	√	√
ES	√	√	√	√	√	√	√
PRE	√	√	√	√	√	√	√

Figure 6.3: Table of dependencies between transformations

The different conclusions are:

- Constant Folding does not influence Elimination with Signs.
- Common Subexpressions Elimination does not influence Elimination with Signs.
- Copy Propagation does not influence Elimination with Signs nor Precalculation.
- Code Motion does not influence Copy Propagation and Dead Code Elimination.

[24, 25] have four transformations in common with the compiler framework implemented in this thesis: Dead Code Elimination (abbreviated DCE as well), Copy Propagation (abbreviated CPP), Code Motion (abbreviated ICM), and Constant Folding (abbreviated CTP for Constant Propagation). Their conclusions about the interactions between these transformations are the same that the ones presented here, except for the three pairs (CPP → DCE), (CPP → CTP) and (ICM → CTP), where they conclude on no interactions while the study made in this thesis concluded on the existence of dependencies. As it is the *existence* of the interactions that must be discussed, the remaining of this thesis will be in favor

of the results *from* the study made in the previous section, because some explicit counter-examples have been provided to show the veracity of these results. The probable reason of this difference between these results and the ones from [24, 25] must come from the possible differences in the definition of the Constant Folding transformation and Dead Code Elimination, where faint variables may not have been considered.

6.2.3 Impact on metrics' calculation

As seen in the previous part, some transformations may not influence the results of others. Thus, it should be the same with metrics, as they are supposed to reflect the effect of the transformations they are representing.

An experimental study has been made in order to get some insights about how the metrics are reacting when a specific transformation is used. An independent Dependency Module has been designed in that aim. This module is used in conjunction with the benchmark suite. During the computation of an order of optimizations (defined by a regular expression), whenever a transformation is called, the Dependency Module gets the different metrics before and after the transformation, and compare the values. If the transformation has made the metric change, a special counter, corresponding to this transformation and this metric, is incremented.

The results of these tests can be seen in the following table (Figure 6.4).

Metrics / Transf.	CF	CSE	CP	DCE	CM	ES	PRE
CF	1.00	0.02	0.07	0.88	0.10	-	0.57
CSE	0.35	1.00	0.50	-	0.10	-	-
CP	0.41	0.01	1.00	-	0.34	-	-
DCE	0.21	0.08	0.01	1.00	0.33	-	0.12
CM	0.96	0.20	-	-	1.00	0.70	0.12
ES	0.76	0.03	-	0.22	-	1.00	0.62
PRE	0.40	0.05	0.01	0.17	0.15	0.08	1.00

Figure 6.4: Table representing the frequency in which a metric's value changed with a specific transformation's successful call

From these results, it is possible to see that the results from the analysis made previously in Section 6.2.2 hold, as the metrics related to transformations that should not be influenced never changed (i.e the frequency of the metrics' changes

is equal to 0).

It can be pointed out that, on this experiment, several pairs give unexpected results: (CP → DCE), (CSE → DCE), (CSE → PRE), (DCE → ES), (ES → CM) and (ES → CP). The frequencies for these pairs are equal to 0, while the analysis made previously showed that the transformation applied could potentially modify the other transformation's results, using examples. The reasons for these values could be the potential over-approximation introduced in the metric calculation (for Common Subexpressions Elimination), or most likely the fact that the situations described in the examples never occurred.

It is now relatively safe to update the metric-based phase-ordering mechanism by avoiding the useless re-computation of metrics that are not changed by specific transformations calls. A special version of the phase-ordering algorithm is used where, after each transformation, the metrics that were not influenced by a previously called transformation are not updated.

6.2.4 Comparison

This last part deals with the comparison of the results of the phase-ordering mechanism with and without the mechanism to avoid useless re-computation of metrics' values. The fact that the Precalculation Process is not directly called after each transformation but associated to a metric is also a parameter that changed between the two approaches.

An interesting point of the dependencies analysis is to note that, though Precalculation is often influenced by other transformations, it is only disabled in some cases (for Common Subexpressions Elimination and Dead Code Elimination), so it does not have to be called after these transformations in the mechanism using the classic update of the metrics.

Figure 6.5 and 6.6 shows the comparison of the time spent in the optimization process between the normal update of the metrics and the new approach, for both the case where the metrics' results are re-used for the transformations calls or not.

Again, Bench Program 10 is not shown in these graphs, but the complete tables of data can be seen in Appendix A.4. The main insight that can be deducted from these graphs is that the new mechanism is globally not improving the optimization time. The two main causes are:

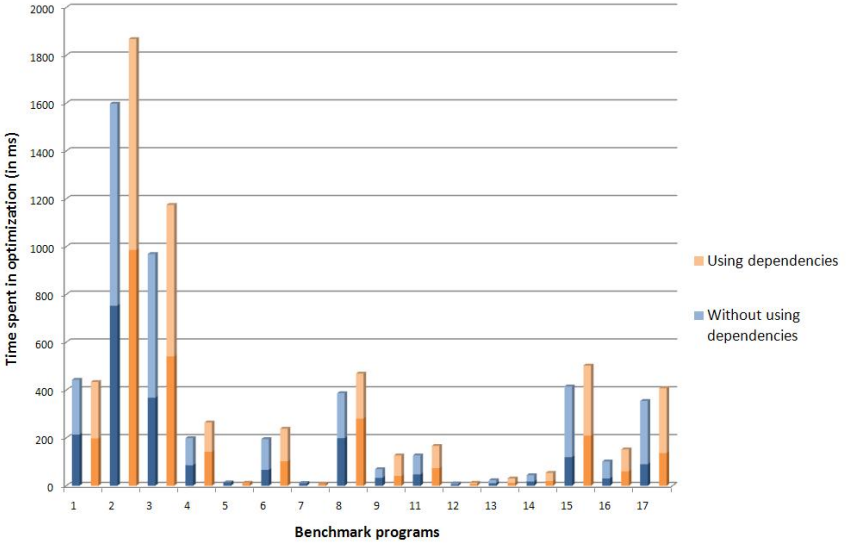


Figure 6.5: Comparison of the optimization time without reuse of the metric's results

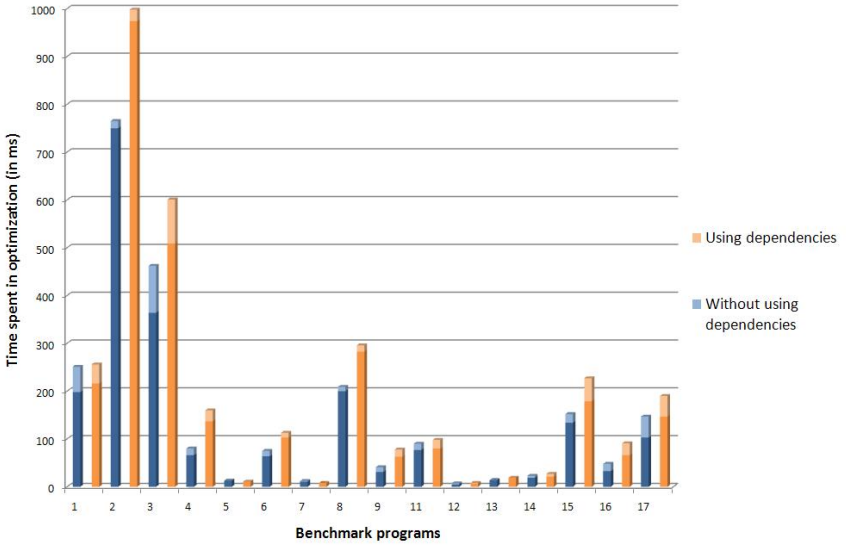


Figure 6.6: Comparison of the optimization time with reuse of the metric's results

- The metric for Precalculation has to be calculated after almost every phases, which increases the metrics' calculation time. This can be compensated by the fact that in the previous approach, Precalculation was called after some of the transformations (though not after all anymore) and calculating the metric takes globally the same amount of time as calling the transformation, since no analysis is involved. With this new mechanism, it must also be added the time to call the transformation. Then the speed-up resulting from the non re-computation of the metrics may be not important enough to make up for this overhead.
- The fact that Precalculation is automatically applied after some phases in the previous version of the phase-ordering mechanism can enable opportunities that are not enabled in this new mechanism. Thus sometimes more transformations calls are needed in the compilation process with this new version.

The results with this new version of the metrics' update are not very satisfying. However, this new mechanism could be more interesting if more transformations were used, as there would be more cases where there is no interaction, improving the speed-up. An even bigger experiment could be made to affine the different probabilities and integrate it in the metrics' calculation mechanism for example, is a similar way as what has been done in [11].

Chapter 7

Evolution of the phase-ordering algorithm

Now that the phase-ordering mechanism has been set up in some specific conditions, it is interesting to investigate how to extend this very mechanism to other goals. This chapter consists first in describing the different goals in program optimization. Then an analysis of the effects of the transformations is performed. The results of this analysis are finally used in an attempt to adapt the phase-ordering mechanism to other goals than speed.

7.1 Goals when optimizing a program

As explained briefly in Section 2, optimizing compilers may have several goals. The overall aim of such a compiler is to "produce code that is more efficient than the obvious code. [...] The compiler must be effective in improving the performance of many input programs." ([1]). The most common requirements are:

- **the speed of the program execution:** it is the "usual" parameter users want to improve using optimizing compiler. The aim here is to reduce the

running time of a program while of course getting the same behavior. It is the parameter that have been used throughout all the experiments of this thesis until now.

- **the size of the generated code:** as embedded applications are more and more common, some users need to shorten their programs, in order to minimize the amount of memory occupied.

- **the power consumed by a program:** because of the growth of portable computers, the power consumption of a program is becoming a more and more important parameter. These mobile devices have in general a limited power supply, so the applications running on them must consume the least possible energy.

Taking the user's need into account is important before compiling a program. Indeed, in embedded applications for example, an optimization can often improve degrade one aspect (e.g size) while improving another one (e.g speed) ([21]), so knowing the goal of the optimization before performing it is necessary.

7.2 Effects of the transformations

The main topic in this section is to investigate the different effects of the transformation according to the different goals defined in Section 7.1. On these three goals, only two of them will be considered: speed and size of the generated program. Indeed, the measure of power consumption is inaccessible in this thesis. Intensive experiments have already been conducted on this topic ([7, 3]), and the integration of such results in this phase-ordering mechanism is left for future work, as explained in Section 9.

The first subsection deals with a theoretical analysis of the effects of different transformations, and aims at finding the potential improvements that these transformations may perform either on the speed or on the size of the program. Again, for the sake of readability, only the part concerning Constant Folding is shown in this section. The remaining transformations can be found in Appendix C.2. The second subsection introduces an experimental study in order to evaluate the results found in the first subsection.

7.2.1 Theory

In this part, the effects of Constant Folding are considered in order to establish whether the transformation is improving the speed or the size of the input program. As the Precalculation Process is applied after some transformations in the metric-based phase-ordering algorithm, a concluding remark is taking the Precalculation into account to evaluate the overall effect of the sequence of the two transformations in that case.

Finally all the effects of the transformations considered are summarized in the last part of this subsection.

7.2.1.1 Effects of Constant Folding

The transformation to be analyzed is the Constant Folding transformation, described in Section 3.4.1.

Functional effect:

The aim of this optimization is to look for all variables assignments with constant values on the right side, and to replace further in the program the variables with their constant value.

Consequences on the program:

When replacing a variable's occurrences by its constant value, the variable itself is less and less used. In fact, most of the time, all the occurrences of the variable between the first definition and the next assignment to this variable are replaced by a constant value, which makes the variable becomes dead at this point of the program. However, the transformation does not remove the dead assignments created.

⇒ Conclusion:

- **Size** ⊘**:** The effect of Constant Folding on the size of the program varies depending on the variables replaced. Indeed, replacing a variable with a long name can improve the size of the program, while replacing a one-letter variable with `true` or `false` can degrade the size for example.
- **Speed** ⊕**:** A successful Constant Folding call will improve the speed

of the program, as runtime variable evaluations are replaced by simple access to constant values.

* **Associated with Precalculation:** Constant Folding will still improve **speed**, while the effect on the **size** are still impossible to predict, but there is more chance that Precalculation improves the size, as Constant Folding may create constant expressions to be computed.

7.2.1.2 Summary

The effects of the transformations considered in this thesis are summarized in Figure 7.1. For each case, a $\oplus$ means that the transformation is very likely to improve the size or speed, a $\ominus$ means it is likely to degrade the size or speed, and a $\oslash$ means that the effects can vary depending on some parameters, such as the length of the variables used for Constant Folding, or the number of expressions replaced for Common Subexpressions Elimination.

	Size	Speed
CP	$\oplus$	$\oplus$
ES	$\oslash$	$\oplus$
CSE	$\oslash$	$\oplus$
DCE	$\oplus$	$\oplus$
CM	$\ominus$	$\oplus$
CF	$\oslash$	$\oplus$

Figure 7.1: Effects of transformations

This study points out that all the transformations considered should improve the speed of the program, characterized in this thesis by the number of instructions executed. On the other hand, all are not always improving the size of the program. Some, like Code Motion, are even very likely to degrade it. For others, like Common Subexpressions or Elimination of Signs, it depends on some properties of the input program.

7.2.2 Experimental results

An experimental study have been conducted using the benchmark suite in order to get some insights about the effects of the transformations on the size and the speed of the program.

Figure 7.2 (respectively 7.3) represents the percentage of transformation calls improving or degrading the size of the program (respectively the speed of the program, measured by the number of instructions executed).

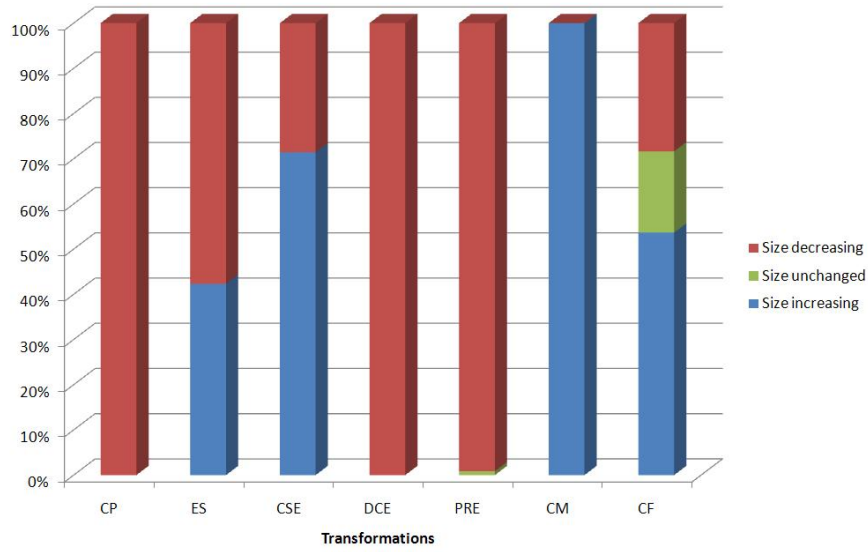


Figure 7.2: Percentage of transformation calls improving/degrading the size

These measures have been made using the regular-expression based approach in order to run a lot of different orders of optimization phases on the benchmark programs. Then, after each transformation call in each regular expression, the relative improvement of the size or speed of the program has been measured using the following coefficient:

$$N = \frac{\mathcal{S}_{before} - \mathcal{S}_{after}}{\mathcal{S}_{before}}$$

where $\mathcal{S}$ is either the size of the program or the number of instruction executed.

These figures show that:

- the speed of the program is never degraded except for a very few percentage of Code Motion calls. The cause of this degradation is very likely to be,

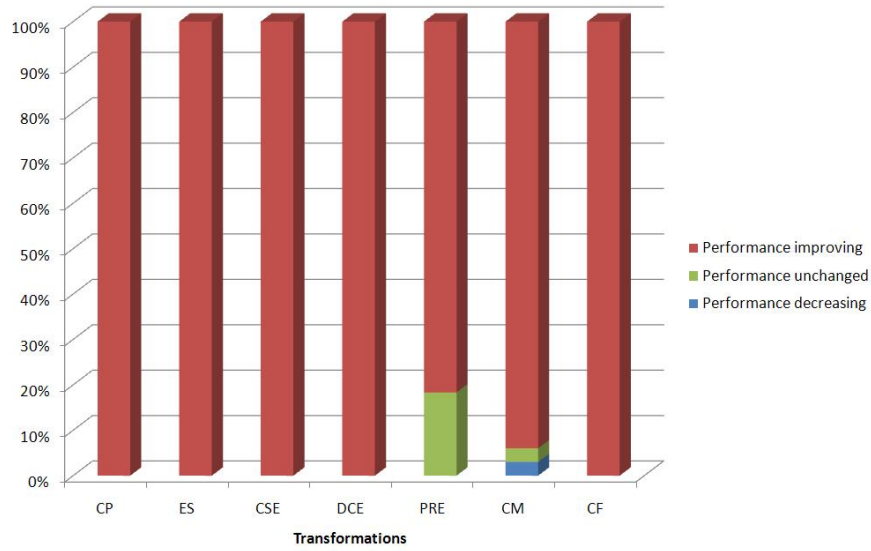


Figure 7.3: Percentage of transformation calls improving/degrading the speed

as mentioned in the theoretical analysis of the effects of Code Motion (in Appendix C.2), the existence of a loop containing invariants and which is only executed once. Otherwise, as expected, all the other transformations are improving the speed of the program (though sometimes Precalculation is not improving, probably only removing useless `skip` statements).

- only two transformations (and Precalculation) are always improving the size of the program: Dead Code Elimination and Copy Propagation. Elimination with Signs improved the size of the program more often than not, but the other three transformations are very likely to degrade it, especially Code Motion which never improved it.

This corroborates the previous analysis, and gives some useful insights on the proportion in which Constant Folding, Elimination with Signs and Common Subexpressions Elimination can improve or degrade the size of the program.

Figure 7.4 and 7.5 shows the average relative improvement of the transformations on the size and the speed of the program.

These figures show the importance of using Dead Code Elimination in both case (speed or size improvement), and permit to establish some comparison between the different transformations.

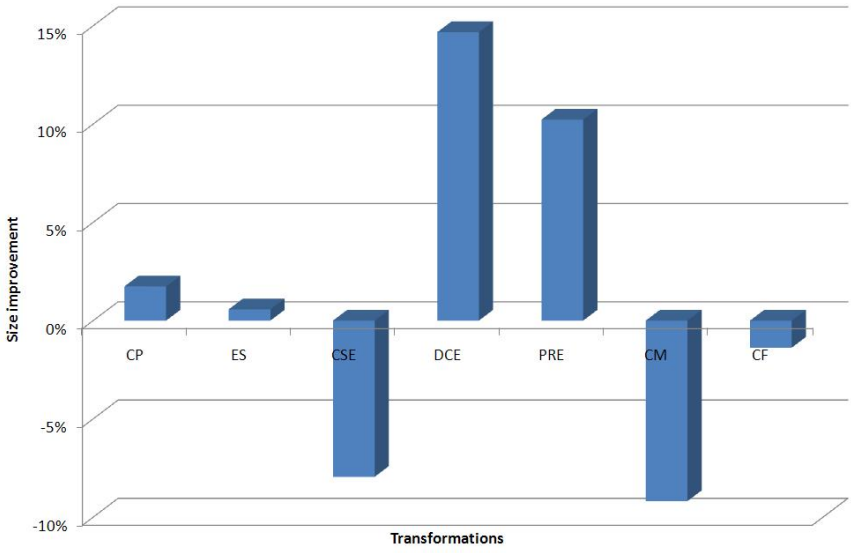


Figure 7.4: Average improvement of transformations on the size of the program

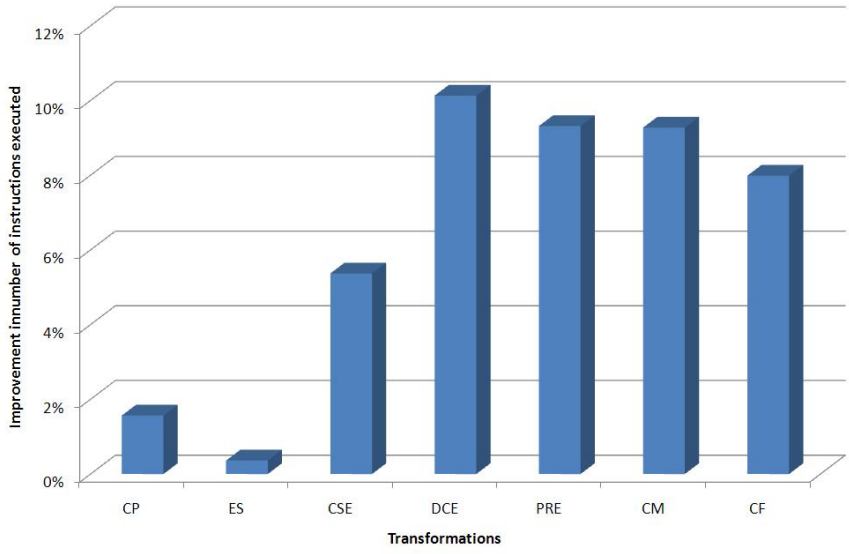


Figure 7.5: Average improvement of transformations on the speed of the program

For example, it shows that Copy Propagation should be of better use for reducing the size than Constant Folding or Common Subexpressions Elimination, but Dead Code Elimination is likely to be more efficient than Copy Propagation. Thanks to these insights, a new mechanism for the comparison of the metrics can be designed, using coefficients to weight the different metrics, in order to improve the phase-ordering mechanism.

7.3 Consequences on the metrics' comparison

As it has been shown in the previous section, optimizing for size is not as easy as optimizing for speed with the transformations considered in this thesis. Indeed, while all these transformations are improving the speed of the program, it is not the same when the goal set is to optimize the size of the program. In order to consider an optimization process that aims at reducing the size of the program, the insights gained in the previous sections can be used to design two mechanisms: a mechanism to detect non-improving transformation calls, and the use of weights when comparing the metrics to know which transformation to choose next.

7.3.1 Detection of non-improving transformation

As some transformations are not always improving the size of the program, a mechanism allowing the detection of non-improving transformation calls has been designed in the phase-ordering algorithm.

The basic idea is to calculate the size of an instance of the program before a transformation call, and compare it with the size after the transformation call. Then if the size increased, the transformation is blacklisted and the old version of the program is taken for the remaining optimization process.

However, some specific cases must be considered. Indeed, some transformations may increase the size of the program at first glance, but enables other transformations to be applied such as the size globally decreases. Consequently, using the informations gathered in Section 6.2.2 about the interactions between the transformations, some transformations are treated specifically:

1. **Constant Folding:** in the case Constant Folding makes the original size of the program decrease, the Phase Manager takes a look at the met-

ric DCE. Indeed, Dead Code Elimination is often enabled by Constant Folding, so there is a chance that applying Dead Code Elimination after Constant Folding is going to be beneficial. So if the metric DCE increases, that means that Dead Code Elimination has been enabled by Constant Folding, and in that case Dead Code Elimination is applied on the program. If the size after Dead Code Elimination is smaller than the original size (before Constant Folding), then the optimization process continues normally, otherwise Constant Folding is blacklisted and the original program (before Constant Folding) is used when resuming the process.

2. **Code Motion:** Code Motion has been shown in the previous section to degrade the size of the program. However, there are some cases where it may be beneficial, combined with Constant Folding and Dead Code Elimination: indeed, the pre-header condition can often be simplified and the pre-header itself can be removed. So in the case Code Motion is applied (and makes the size increase), Constant Folding is considered in a similar way as Dead Code Elimination above. If this still does not improve the size, another call of Constant Folding (because Precalculation, called after the first Constant Folding may have removed the pre-header and enabled Constant Folding again) is made combined with Dead Code Elimination. If this still degrades the size of the program, then Code Motion is blacklisted and the original program is used in the rest of the optimization process.

Calling other transformations without being sure it will be beneficial can increase the compilation time, so it could be interesting to affine the choice of the order in which the transformations will be considered. The next part introduces some coefficients used to weight the different metrics in order to reduce the compilation time.

7.3.2 Use of weights in phase-ordering mechanism

This second part deals with the mechanism that allows to weight the metrics' values according to the probability they have to improve the program. This can be used to help the phase-ordering mechanism to make a better choice of transformation depending on the goal specified by the user.

The previous section considered transformations that are not improving the size of the program. A good way to reduce the optimization time is to applied first the transformations that are unlikely to degrade the size of the program, and let the problematic transformations be called at the end. In order to classify the transformations by their probability to improve the size but still consider the

metric associated with them, a system of weighting has been designed. Whenever a metric is calculated, it is multiplied by a coefficient that represents the weight of the transformation.

These weights have been defined according to the results of Section 7.2.2: the most often a transformation is improving the size in the previous experiments, the higher will be the weight of the metric associated to this transformation.

A rough estimation of potentially good weights designed from Figure 7.4 is:

- Copy Propagation: 1.3
- Elimination with Signs: 1
- Common Subexpressions Elimination: 0.3
- Dead Code Elimination: 2
- Code Motion: 0.1
- Constant Folding: 0.7

These values have been chosen as an example, but of course a more in-depth choice should be realized for optimal performance.

7.3.3 Evaluation

This last part deals with the evaluation of the phase-ordering mechanism aiming at reducing the size of a program. Figure 7.6 shows the size of the optimized benchmark programs in three cases:

- using the benchmark suite and choosing the best regular expression according to the size of the program and then the optimization time.
- the classic metric-based phase-ordering mechanism evaluated in Section 6.1, with reuse of the results from the metrics' calculation.
- the metric-based phase-ordering using the two mechanisms from the previous sections aiming at improving the size of the optimized program.

The size of the optimized benchmark programs are almost always the same, except for two benchmark programs: in one of them (BenchPrg1), the best regular

	Best regular expression	Metric-based without weights	Metric-based with weights
BenchPrg1	331	365	331
BenchPrg2	1781	1781	1781
BenchPrg3	723	698	698
BenchPrg4	431	431	431
BenchPrg5	536	536	536
BenchPrg6	438	438	438
BenchPrg7	582	582	582
BenchPrg8	411	411	411
BenchPrg9	87	87	87
BenchPrg10	3731	3731	3731
BenchPrg11	192	192	192
BenchPrg12	89	89	89
BenchPrg13	111	111	111
BenchPrg14	62	62	62
BenchPrg15	568	568	568
BenchPrg16	185	185	185
BenchPrg17	402	402	402

Figure 7.6: Size of the optimized benchmark programs

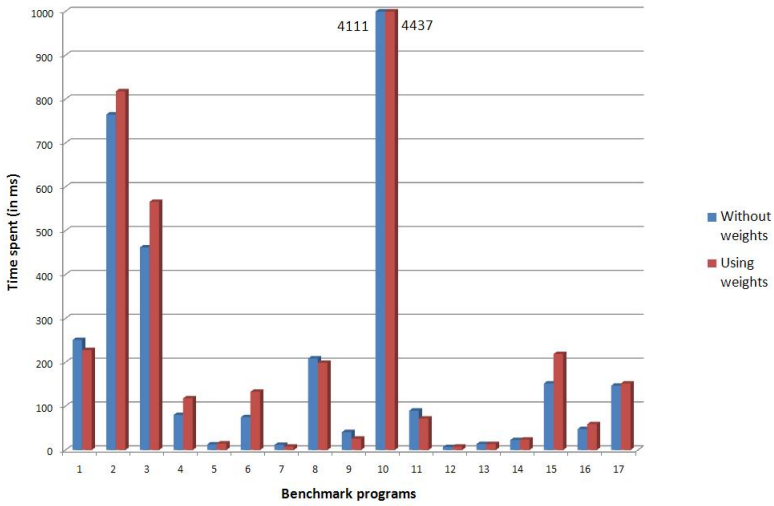


Figure 7.7: Optimization time for metric-based phase-ordering

expression compute a program with a smaller size than the classic metric-based algorithm, as in the other (BenchPrg3), the classic metric-based algorithm is performing better than the best regular expression in terms of size. In both cases, the new mechanism presented in this section is producing a program with the smallest size.

Finally, introducing the detection of non-improving transformation calls increased slightly the optimization time, compared to the metrics' choice without any weight, as illustrated in Figure 7.7.

The corresponding table of data can be found in Appendix A.5. This overhead is compensated by the fact that the mechanism using this detection is likely to produce better results for the size of the program, for example for BenchPrg 1.

Thus, this new version of the phase-ordering algorithm using weights and a non-improving transformation calls detection mechanism is an interesting improvement and proved to be efficient on these benchmark programs. However it requires some knowledge about the interactions between the transformations in order to spot the good sequences that can be efficient while single transformation calls are not.

Chapter 8

Design and implementation

This chapter deals with the design and the implementation of the Phase Manager in Java and all the different auxiliary modules that are used for the manager, the benchmarks and the metrics. After considering the implementation language chosen, the second section contains references to the main classes and methods involved in the optimization part of the compiler framework, while the last section addresses the different implementation issues that occurred during this thesis.

8.1 Implementation language

The WHILE compiler is written using the Java 1.4 programming language. The main advantage of this language is that it is an object-oriented language, which allows an object representation of the different elements of the parse tree representing the intermediate representation of a program (see Section 3.2.1).

8.2 Main classes

This section introduces the main Java classes that were implemented for the compiler framework, with an emphasis on the Phase Manager and the other modules involved in the optimization process.

The frontend of the compiler (the lexical analyzer and the parser) is defined in the *Parse* package. The lexical analyzer has been generated using a tool JLex, which tokenizes the input program into different terminal tokens. The parser is constructed using the Java CUP tool. It takes as input a description of how the program should be parsed and which Java objects should be created depending on the situation.
As a result, the parse tree is composed by Java objects representing the syntax of the program. These classes are contained in the package Absyn, and are subclasses of the Absyn.Absyn class. The overall program is constructed as an instance of the Prg class, which has two fields, a DecList object that represents the variable declaration, and a Stm object that represents the body of the program. All statements in the program are instances of the Stm class (in fact, they are instances of classes that extends the Stm class, like IfStm for a if-statement, or WhileStm for a while-statement...), and all expressions are extending the Exp class (VarExp, ConstExp...).

The Phase Manager itself has been implemented in a class called *PhaseManager* of the *optimizer.manager* package. In this class are implemented all the methods that are used to interprete the regular expressions and launch the benchmarks as well as the calculation of the different metrics and the algorithm to use the metric to optimize the program, or the method *updateUDandDUchains(...)* to compute the UD- and DU- chains for example. The two main method are:

- The method *optimizeUsingMetrics(...)* that performs the dynamic phase-ordering algorithm. This method calculates the different metrics for all the optimizations available.
- The method *launchBenchmark(...)* that handles the benchmark tests. This method uses the Regular Expression Generator to generates regular expressions, then for each program (specified by a file), applies each regular expressions and forwards all the results to the Record Analyser.

In the *optimizer.manager* package are also defined two other utility classes: the class *DependenciesUtilities* and the class *EffectsUtilities* that contain various methods to evaluate respectively the dependencies between the transformations (Section 6.2.2) and the effects of these transformations (Section 7.2).

The Regular Expression Generator is defined in the *regExp.RegExpGenerator* class. Each module is represented by one method, all grouped together in the main method called *generateRegExps(...)*. This method calls all the Generator's modules and creates an array of regular expressions that can then be used in the benchmark suite.

The Record Analyzer is implemented in the *RecordAnalyser* class of the *optimizer.analyser* package. It gets *Record* objects from the Phase Manager as input, and analyzes them in the method *analyseBench(...)*, which updates the different *java.util.HashMap* objects containing the different rankings.

Finally, the Optimizer class from the *optimizer.optimizer* package is the class containing all the data-flow analyses and transformations implemented to work with the Phase Manager. Each transformation can be called using one method, and two different analysis algorithms are available (the MFP algorithm and the Abstract Worklist Algorithm).

8.3 Implementation issues

During this thesis, several implementation issues made this work challenging. First, producing code that will always generate the correct output for several different sequences of optimizations is difficult. Even the task of implementing a conventional compiler that has to produce correct output for several predefined optimization phase sequences is not easy at all. So producing code that will always execute correctly for hundreds of different sequences of optimizations is, in contrast, a severe stress test.

Making sure that all the attempted sequences will produce valid code required backtracking a lot of different errors that were not discovered previously when implementing the optimizations in the WHILE compiler. These errors made the quantity of benchmark runs increase severely, which can be represented as more than a hundred hours spent on the benchmarks.

However, the final version of the Optimizer and the Phase Manager implementation have produced code that shown very good results, where no errors were discovered again, increasing the confidence that can be put in the correctness of this final implementation.

Chapter 9

Future work and perspectives

There is much future work to consider on this novel approach of metric-based phase-ordering in optimizing compilers. This chapter introduces a variety of improvements that are planned to be investigated in the future. The first section deals with the design of new metrics and the extension of the WHILE language; the second section concerns the study of new analyses and transformations; while the third section introduces the possible integration of static power and performance models to have wider insights on the effects of the transformations applied during the optimization. Finally, the last section considers the experimentation setup.

9.1 Designing new metrics and extending the WHILE language

A first improvement to the actual mechanism would be the creation of new metrics. It would be interesting to design metrics that represent not only the potential effects of individual transformations, but also sequences of transformations that have been shown to be efficient. Considering the Constant Folding transformation, Section 6.2.2 about the interactions between the transformations shows that it has a high probability of enabling Dead Code Elimination.

This insight has already been used in the previous section when designing a mechanism to detect non-improving transformation calls for size-aimed optimization. Designing a metric that considers the combined action of both Constant Folding and Dead Code Elimination may thus improve the optimization process.
Others sequences of optimization phases can also be considered, as Code Motion followed Constant Folding for example. Research on a better understanding of the interactions between the transformations is still going on, and may give useful insights on the sequences of optimization phases on which a metric would be profitable.

The WHILE language considered in this thesis is a simple imperative language containing most of the interesting features of other well-known languages like C. An interesting future work would be to extend this language with other types of statements, from the `switch` statement to the `goto` and `break` statements. Some mechanisms used in the propagation algorithm, such as the visiting graph, would need to be modified to work with these new statements.
Another improvement to the analysis of the WHILE language would be to consider interprocedural analyses instead of only intraprocedural analyses. Again, the visiting graph would need to be extended in order to take function calls into account.

Finally, the work done in this thesis was planned to be integrated in the Microsoft Phoenix framework. Phoenix is a codename for Microsoft's next-generation, state-of-the-art infrastructure for program analysis and transformation. It aims at creating an industry leading compilation and tools framework. Implementing the propagation algorithm in Phoenix has not been attempted due to lack of time. It would have required to adapt the algorithm to the Phoenix Intermediate Representation and the programming language used, and to consider the transformations available. Thus, the design and implementation of the propagation algorithm and the metric-based phase-ordering mechanism is left for a very interesting future work as well.

9.2 Adding analyses and transformations

Additionally, the set of optimization phases could be increased. Seven transformations (including Precalculation) has been considered in this thesis. Adding new transformations could enable other interesting interactions, and raise new challenges when designing the metrics. Previous work could also be used to generate new metrics: in [4] for example, Click and Cooper combines Constant Propagation (equivalent to the Constant Folding used in this thesis) and Un-

reachable Code Elimination, which aims at removing statements that will never be executed.

However, adding new transformations may require new analyses to be used in the propagation algorithm. A challenging future work would be to find out what type of analyses can run this algorithm. Then, a model could be established to characterize these analyses and the propagation algorithm could be shaped in a more abstract way. A good start would be to use constraints in the version for bit-vector analyses in order to group Constant Propagation and Elimination with Signs with the bit-vector analyses in a single and general version of the propagation algorithm.

9.3 Integrating power and performance models

Another area of future work is to study the effects of the transformations in a more detailed way. When optimizing for size, the phase-ordering mechanism has been modified in order to detect non-improving transformations. Adding such a mechanism when optimizing for speed was not necessary in this thesis, as all the optimizations considered have been shown to be beneficial in terms of speed. However, in the case other transformations would be added, a runtime performance evaluation of the optimized program could be useful.
Performance evaluation is a very important factor in optimizing compilation, as has been shown in Section 2. It would then be interesting to investigate the static performance evaluation functions currently used in the literature and in the research area: indeed, performance models that predict running time could greatly reduce the cost of evaluating the effect of a transformation call, and thus permit the design of a runtime detection of non-improving call, as was done for size.

The issue of the termination of the metric-based phase-ordering algorithm has been addressed in Section 5.5.2. This was based on the assumption that a sequence of transformations could not be applied on a program P and output the same program P, making a loop in the optimization process. The use of a mechanism that detects non-improving calls would solve this issue, as the program would be always improving, finally reaching a fixed point (because the execution time has a lower bound set to 0 sec.). Even without this mechanism, the situation described above is unlikely to appear. However, another mechanism could be designed to detect this kind of looping in the optimization process and solve it by blacklisting the transformations responsible for it.

Finally, the integration of power evaluation models could also be used in or-

der to provide the user the choice to optimize for power consumption. The optimization process would also use the same mechanisms as when optimizing for size. For example, Kandemir et al. use in [7] a power estimation tool in their experiments, called *SimplePower*.

9.4 On the experimentation

Last but not least, the set of benchmark programs considered in the experiments could be increased. Indeed, several key insights in this thesis came from the experimentation made using the implementation of the Phase Manager. Thus, performing experiments of larger scale would obviously give more interesting results.

Chapter 10

Conclusion

The design of optimizing compilers has always been a difficult task. One of the oldest problems is to find the best order of optimization phases to be applied in order to produce optimal code. This issue, called the *phase-ordering problem*, has been investigated since many years, as applying a fixed sequence of optimization phases is widely acknowledged not to produce optimal code for every applications.

Optimizing compilation for high-performance computing or embedded applications involves iterative compilers that generate various sequences of optimization phases and evaluate them in order to find the best ordering. This approach takes a considerable amount of time. On the other hand, classic command-line compilers only provide the user with some specific options that applies optimization phases to all programs in one fixed order.

This thesis describes a novel approach to the phase-ordering problem. A new entity, the Phase Manager, is calculating coefficients, called *metrics*, to evaluate where the different transformations can be applied, and choose dynamically the order of optimizations during the optimization process. These metrics take into account the intermediate representation of the program, so that a specific sequence of optimizations is computed for each program.

The Phase Manager has been implemented in Java for a simple but complete

imperative language and for seven transformations using data-flow analyses. The metrics use a new algorithm that has been shown to solve these analyses faster than the classical algorithms. This metric-based approach has been compared to the results from a benchmark suite that generates a large amount of optimization sequences using regular expressions. The new approach has been shown to optimize the benchmark programs in a globally faster time than the *best* regular expressions for each program.

This thesis also considered the interactions between the transformations to try to improve the metric-based mechanism. The new technique used has not shown significant improvements compared to the first version of the metric-based phase-ordering mechanism.

Finally, the phase-ordering algorithm has been extended to consider an optimization process aiming at reducing the size of the program. This version has been evaluated and shown to produce a smaller (or equally small) code than the benchmark suite or the first version of the algorithm.

Several perspectives of future work have already been considered, but the potential of this new approach to the phase-ordering problem has been proved in this thesis. In particular, the new algorithm used to evaluate the data-flow analyses showed a good potential to decrease the time spent in solving data-flow equations.

Appendix A

Benchmark results

In this appendix are shown the different benchmark results that are not part of the other chapters. This includes:

1. The list of regular expressions used in the benchmarks
2. The tables of data from the metric-based phase-ordering evaluation
3. The data from the regular expressions reaching the minimum number of executed instructions
4. The tables of data for the metrics' calculation using or not using the dependencies analysis
5. The tables of data for the optimization aiming at reducing the size of the program

A.1 List of regular expressions used

```
xxxxxxxxxxxxxxx List of the regular expressions used in optimisation xxxxxxxxxxxxxxx

Reg. exp. 0: PRE.(CSE.CP.PRE)*.(CF.PRE)*.DCE.PRE.CM.(CF.ES.PRE)*
Reg. exp. 1: PRE.(CF.PRE)*.(CSE.CP.PRE)*.DCE.PRE.CM.(CF.ES.PRE)*
Reg. exp. 2: PRE.(CSE.CP.PRE)*.(CF.PRE)*.(CF.ES.PRE)*.DCE.PRE.CM
```

```
Reg. exp.  3: PRE.(CF.PRE)*.(CSE.CP.PRE)*.(CF.ES.PRE)*.DCE.PRE.CM
Reg. exp.  4: ES.PRE.DCE.CSE.CM.CF.CP
Reg. exp.  5: (ES.PRE.DCE.CSE.CM.CF.CP)*
Reg. exp.  6: ES.PRE.DCE.CSE.CM.CF.CP
Reg. exp.  7: ES.PRE.DCE.CSE.CM.CF.CP
Reg. exp.  8: (ES.DCE.PRE.CSE.CF.CP.CM)*
Reg. exp.  9: (ES.DCE.PRE.CSE.CF.CP.CM)*
Reg. exp. 10: ES.DCE.PRE.CSE.CF.CP.CM
Reg. exp. 11: ES.DCE.PRE.CSE.CF.CP.CM
Reg. exp. 12: (ES.DCE.PRE.CSE.CF.CP.CM)*
Reg. exp. 13: ES.DCE.PRE.CSE.CF.CP.CM
Reg. exp. 14: ES.DCE.PRE.CSE.CF.CP.CM
Reg. exp. 15: (ES.DCE.PRE.CSE.CF.CP.CM)*
Reg. exp. 16: (ES.DCE.PRE.CSE.CF.CP.CM)*
Reg. exp. 17: ES.DCE.PRE.CSE.CF.CP.CM
Reg. exp. 18: (ES.DCE.PRE.CSE.CF.CP.CM)*
Reg. exp. 19: ES.DCE.PRE.CSE.CF.CP.CM
Reg. exp. 20: ES.DCE.PRE.CSE.CF.CP.CM
Reg. exp. 21: (ES.DCE.PRE.CSE.CF.CP.CM)*
Reg. exp. 22: ES.DCE.PRE.CSE.CF.CP.CM
Reg. exp. 23: ES.DCE.PRE.CSE.CM.CF.CP
Reg. exp. 24: (CP.DCE.CSE.CM.PRE.CF.ES)*
Reg. exp. 25: CP.DCE.CSE.CM.PRE.CF.ES
Reg. exp. 26: (CP.DCE.CSE.CM.PRE.CF.ES)*
Reg. exp. 27: (CP.DCE.CSE.CM.PRE.CF.ES)*
Reg. exp. 28: (CP.DCE.CSE.CM.PRE.CF.ES)*
Reg. exp. 29: (CP.DCE.CSE.CM.PRE.CF.ES)*
Reg. exp. 30: (CP.DCE.CSE.CM.PRE.CF.ES)*
Reg. exp. 31: CP.DCE.CSE.CM.PRE.CF.ES
Reg. exp. 32: CP.DCE.CSE.CM.PRE.CF.ES
Reg. exp. 33: (CP.DCE.CSE.CM.PRE.CF.ES)*
Reg. exp. 34: (CP.DCE.CSE.CM.PRE.CF.ES)*
Reg. exp. 35: CP.DCE.CSE.CM.PRE.CF.ES
Reg. exp. 36: CP.DCE.CSE.CM.PRE.CF.ES
Reg. exp. 37: (CP.DCE.CSE.CM.PRE.CF.ES)*
Reg. exp. 38: CP.DCE.CSE.CM.PRE.CF.ES
Reg. exp. 39: CP.CF.CSE.CM.ES.DCE.PRE
Reg. exp. 40: (DCE.CSE.CF.CM.ES.CP.PRE)*
Reg. exp. 41: (DCE.CSE.CF.CM.ES.CP.PRE)*
Reg. exp. 42: (DCE.CSE.CF.CM.ES.CP.PRE)*
Reg. exp. 43: (DCE.CSE.CF.CM.ES.CP.PRE)*
Reg. exp. 44: DCE.CSE.CF.CM.ES.CP.PRE
Reg. exp. 45: (DCE.CSE.CF.CM.ES.CP.PRE)*
Reg. exp. 46: (DCE.CSE.CF.CM.ES.CP.PRE)*
Reg. exp. 47: DCE.CSE.CF.CM.ES.CP.PRE
Reg. exp. 48: (DCE.CSE.CF.CM.ES.CP.PRE)*
Reg. exp. 49: DCE.CSE.CF.CM.ES.CP.PRE
Reg. exp. 50: CM.(((CSE) + (DCE).ES.((DCE)*.CSE.CF.(PRE)*) + ((CP)
        + (ES.(CSE.CM.CM.(CSE)*)*).(PRE)*).CF) + (PRE))*
Reg. exp. 51: (PRE) + (PRE).(CSE)*
Reg. exp. 52: (((DCE)*) + (ES.(CP.(CSE.PRE.PRE.((CM.CSE.DCE.(ES.(CSE)*)*)*)
        + (ES).(PRE) + (CSE).CSE.PRE)*)*))*.CF
Reg. exp. 53: PRE.(CSE.(CSE)*)*.(((PRE.((DCE.(CP)*) + (CP))*) + (CP))*.(PRE)*)*
Reg. exp. 54: CF.(ES)*
Reg. exp. 55: CP.CF.(DCE.CF.(CM)*)*
Reg. exp. 56: ((PRE) + (PRE))*.(PRE.((PRE)*.PRE.(DCE)*)*)*
Reg. exp. 57: (((((ES.PRE.PRE.PRE.(PRE)*)*) + (CF)) + (PRE)) + (CF))*.(PRE)*
Reg. exp. 58: (DCE)*.(PRE.ES.CF.(CF)*.(ES)*)*
Reg. exp. 59: CF.(PRE)*
Reg. exp. 60: (CP)*.(CF.ES.(CM.((CSE.PRE.CSE.(DCE.(CF)*)*)*.PRE.(CP)*)*).(ES.(CF)*.(CSE)*)*)*.ES.CP
Reg. exp. 61: (PRE)*.(CF)*
Reg. exp. 62: (CM)*.CSE.(((CF)*.(DCE)*.(CM) + (((ES)*.(DCE)*) + (DCE)).PRE.ES.CF.(PRE)*) + (PRE))*
Reg. exp. 63: CP.((CF)*.PRE.CF.(PRE)*)*
Reg. exp. 64: (CM)*.CF.(PRE)*
Reg. exp. 65: PRE.CF.CSE.(ES)*
Reg. exp. 66: CF.PRE.(CSE)*
Reg. exp. 67: PRE.PRE.(PRE)*
Reg. exp. 68: CM.ES.((CP) + (CF))*.CSE.(CF)*
Reg. exp. 69: (PRE)*.(CF)*
Reg. exp. 70: PRE.(CM)*
Reg. exp. 71: (ES.CF.PRE.(PRE.(CSE)*)*)*.(PRE.(PRE)*)*
Reg. exp. 72: ES.(((CSE)*) + (ES).((DCE)*.CP.(PRE)*)*)*
Reg. exp. 73: DCE.(CP)*
Reg. exp. 74: ES.(PRE)*
Reg. exp. 75: (CF)*.(CP)*
Reg. exp. 76: CP.((CM) + (CF))*
Reg. exp. 77: (CM) + ((DCE)*.CF.(CF)*).(ES)*.CM.(PRE)*
Reg. exp. 78: ((CSE) + ((PRE.(PRE)*)*)*.CSE.CSE.(CM)*
Reg. exp. 79: CM.((CP) + (CP.(CSE)*))*.(PRE.ES.PRE.DCE.CF.(CP.(CP)*)*.CP.((PRE)*.CF.(ES.CSE)*)*)*
Reg. exp. 80: CF.(CP)*
Reg. exp. 81: (ES)*.CF.(PRE) + (CSE).((CM) + (PRE.(PRE.(CM.(((CSE.CF.PRE.(CP)*)
        + (CM))*)*)*))*.DCE.(PRE)*
Reg. exp. 82: PRE.(DCE)*
Reg. exp. 83: CM.(DCE.(CSE)*)*
Reg. exp. 84: CM.(ES)*
Reg. exp. 85: PRE.CF.(((PRE) + ((PRE)*.PRE.(PRE) + (CF).(CP)*).PRE.(CF.(CSE)
        + (PRE).DCE.CSE.(DCE) + (CM).PRE)*) + (PRE))*
```

```
Reg. exp. 86: CP.CF.((CF) + (PRE))*.(PRE)*
Reg. exp. 87: (CF)*.PRE.(CP)*
Reg. exp. 88: (CP)*.DCE.PRE.(ES.PRE.((((CM.CP.(PRE)*) + (CSE)) + (PRE))*)*
Reg. exp. 89: ES.CM.(PRE.(CM)*)*
Reg. exp. 90: DCE.(PRE.(ES)*)*
Reg. exp. 91: PRE.(CM)*
Reg. exp. 92: DCE.CF.(DCE.(CM)*)*
Reg. exp. 93: (CSE) + ((PRE)*).(CM)*
Reg. exp. 94: (ES)*.(PRE)*
Reg. exp. 95: (CM)*.CP.(CSE)*
Reg. exp. 96: CF.(PRE)*
Reg. exp. 97: CM.(ES)*
Reg. exp. 98: (CM)*.(CM)*
Reg. exp. 99: (PRE)*.(PRE.((CF)*) + (CF).(CF)*)*
Reg. exp. 100: CF.(PRE)*
Reg. exp. 101: CM.(CP)*.(CP)*
Reg. exp. 102: (CP)*.(PRE)*
Reg. exp. 103: CSE.(PRE)*
Reg. exp. 104: CP.((CP)*.(PRE.((CP) + (PRE))*)*)*
Reg. exp. 105: PRE.(PRE)*.(ES)*
Reg. exp. 106: CM.(PRE.CP.(DCE)*) + ((((PRE)*) + ((ES.(CM)*)*)) + (PRE)).((CM)
        + (ES.CP.ES.((CM) + (ES).(DCE.PRE)*)*).PRE)*
Reg. exp. 107: (((CF) + (CF))*) + ((ES) + (PRE)).(ES)*
Reg. exp. 108: CSE.(CSE.(PRE.(CM)*.(PRE.(DCE.(ES)*)*)*)*)*
Reg. exp. 109: (ES)*.(DCE) + (CSE).(CM.(ES) + (CM).(PRE)*)*
Reg. exp. 110: CM.DCE.(ES)*
Reg. exp. 111: (CP) + ((((PRE)*) + (CSE))*).(DCE)*
Reg. exp. 112: (DCE.ES.(PRE)*)*.(ES.(CP)*)*
Reg. exp. 113: CF.(CP.(CM)*)*
Reg. exp. 114: CF.((DCE) + (PRE))*
Reg. exp. 115: CP.(DCE)*
Reg. exp. 116: PRE.((PRE.(ES)*)*.((DCE) + (ES))*)*
Reg. exp. 117: DCE.((CF) + (ES))*.(((CP.((CF)
        + ((PRE) + (PRE)))*)*.PRE.(PRE.(CF.(CP)*)*)*) + (PRE))*
Reg. exp. 118: CSE.DCE.CP.CF.(CM)*
Reg. exp. 119: ((ES) + (PRE))*.((CM) + (CP).CF.(PRE)*)*
Reg. exp. 120: ES.(CM.DCE.(CM.(CF)*)*)*
Reg. exp. 121: CF.CP.PRE.((CP) + (PRE.(DCE)*))*
Reg. exp. 122: (CF)*.(CF)*
Reg. exp. 123: (DCE.ES.PRE.PRE.(CM)*)*.(ES.DCE.(CM.DCE.CP.(ES)*)*)*
Reg. exp. 124: CP.DCE.(PRE)*
Reg. exp. 125: CP.(DCE.PRE.(DCE.(CF.DCE.((ES.(CF)*)*.(ES)*)*)*)*)*
Reg. exp. 126: (CF) + (ES).(PRE)*
Reg. exp. 127: ES.(((CSE)*) + ((CSE) + (CSE)))*.(((((CF) + (((CM) + (CSE.CF.(CM)*)) + (CM)))
        + (ES))*.PRE.(CF)*) + (CM).(CP)*)*
Reg. exp. 128: (((PRE)*) + ((CM)*).CSE.(ES)*)*.(CM.CP.CP.PRE.(CF)*)*
Reg. exp. 129: (CSE)*.(CM.CM.((CSE)*.(PRE)*)*)*
Reg. exp. 130: CF.(CF)*.(CM)*
Reg. exp. 131: PRE.PRE.(CM)*.CM.((CF.(ES)*)*.((PRE) + ((PRE)*))*)*
Reg. exp. 132: CF.(CSE)*
Reg. exp. 133: (PRE)*.(PRE.(CF)*)*
Reg. exp. 134: (((CSE)*.((PRE.PRE.(DCE)*) + ((ES)*))*)*.(ES)*).CF.(ES.(PRE)*)*
Reg. exp. 135: PRE.(CM)*.(PRE)*
Reg. exp. 136: PRE.CSE.(((CF)*) + (((CP)*) + (DCE)).(CM)*)*.(CM.PRE.(CSE)*)*.(CF)*
Reg. exp. 137: CSE.PRE.(PRE)*
Reg. exp. 138: CF.(PRE)*
Reg. exp. 139: (CM)*.PRE.ES.(PRE)*
Reg. exp. 140: PRE.CF.PRE.(ES)*
Reg. exp. 141: CSE.(PRE.(ES)*)*
Reg. exp. 142: PRE.CP.ES.PRE.(CF)*
Reg. exp. 143: DCE.CSE.(PRE)*
Reg. exp. 144: PRE.CP.(CF)*
Reg. exp. 145: (CSE.CM.CP.(PRE)*) + ((CSE) + ((CSE) + ((CSE.CP.(CF)*)
        + ((PRE)*).((CSE)*.(CM)*)*).(PRE)*)).(PRE)*
Reg. exp. 146: (CF)*.(PRE)*
Reg. exp. 147: (CF)*.(ES)*
Reg. exp. 148: CSE.CSE.(PRE.(DCE)*.(CF)*)*
Reg. exp. 149: PRE.(DCE)*
Reg. exp. 150: ((PRE.PRE.(CF.CSE.(PRE)*)*)*) + (ES).(CM.(PRE)*)*
Reg. exp. 151: ES.(ES)*
Reg. exp. 152: CSE.CM.(CP)*
Reg. exp. 153: PRE.(PRE)*
Reg. exp. 154: ES.(((PRE)*.(PRE)*.(CM)*) + (CM))*
Reg. exp. 155: CP.((PRE.(PRE.(ES)*)*) + ((PRE) + (CP)))*
Reg. exp. 156: ((DCE) + (((CF) + ((CP)*.(CSE)*).PRE.CM.((PRE.CSE.CP.CSE.CM.((PRE) + (ES))*)
        + (PRE))*) + (ES))) + (PRE).CF
Reg. exp. 157: PRE.(CP)*
Reg. exp. 158: CP.(PRE)*
Reg. exp. 159: (ES)*.(PRE) + (PRE).(((CSE.PRE.((CF) + (DCE))*.((PRE)*.ES.(CP.PRE.((DCE)
        + (CF).(CM)*)*)*)*)*) + (PRE))*
Reg. exp. 160: CF.((PRE.(ES)*)*.(DCE)*)*
Reg. exp. 161: CSE.((CF)*.PRE.(PRE)*)*
Reg. exp. 162: (DCE)*.(ES)*.(PRE)*
Reg. exp. 163: ES.(PRE)*
Reg. exp. 164: (PRE)*.PRE.(PRE)*
Reg. exp. 165: PRE.(PRE)*
Reg. exp. 166: (PRE)*.(CSE.(CF)*)*
```

```
Reg. exp. 167: (CF)*.(PRE)*
Reg. exp. 168: DCE.((ES) + ((CF) + (CM))) + ((DCE.(CP)*)*).CF.(CF)*.(ES)*
Reg. exp. 169: DCE.DCE.((DCE.CF.((PRE) + ((CM)*))*)*.(CM)*)*
Reg. exp. 170: DCE.DCE.((PRE) + ((CF)*))*
Reg. exp. 171: (DCE.(PRE)*)*.(CM.CF.(PRE) + (PRE).((CF)*.CM.CSE.PRE.PRE.PRE.(CM)*)*.CF.CP.DCE)*
Reg. exp. 172: (DCE)*.CF.(CSE.(DCE)*)*
Reg. exp. 173: CSE.((DCE.(PRE)*) + (DCE))*
Reg. exp. 174: CF.((ES)*) + (CP).CM.(CM.(ES.(PRE.(PRE.(PRE.(PRE)*)*)*)*)*.(DCE) + (CM).(PRE)*)*
Reg. exp. 175: CP.CSE.(PRE)*
Reg. exp. 176: (CF)*.(ES.(CP)*)*.(CP)*.(PRE)*.ES.DCE.(CF.(CSE.(CSE)*)*)*
Reg. exp. 177: CF.CM.(CM)*.(CP)*
Reg. exp. 178: CP.(PRE.(CSE)*)*
Reg. exp. 179: (PRE.(CSE)*.(CP.((ES) + ((DCE)*))*)*) + (DCE).(DCE)*
Reg. exp. 180: CM.(PRE)*
Reg. exp. 181: PRE.((PRE)*.(PRE)*) + ((CM)*).CSE.CM.((CSE.(DCE)*.(ES)*)*.CSE.(CP)*)*
Reg. exp. 182: DCE.PRE.(PRE)*
Reg. exp. 183: (PRE) + ((CM) + (DCE)).(CP)*
Reg. exp. 184: ES.(PRE)*
Reg. exp. 185: ES.PRE.(CM)*
Reg. exp. 186: PRE.(CM.((PRE.((CF) + (PRE))*)*.(CM.(CSE)*)*)*)*
Reg. exp. 187: ((CP) + (CF))*.CSE.((CF) + (PRE.(CSE)*.ES.(((PRE)*)
         + (PRE.(PRE.(PRE)*.(CSE)*)*))*).(PRE)*)*.(ES)*.(PRE.ES)*
Reg. exp. 188: (CF)*.(CSE)*
Reg. exp. 189: PRE.(CM)*
Reg. exp. 190: CF.(PRE)*
Reg. exp. 191: CF.((CP) + (DCE))*
Reg. exp. 192: PRE.(PRE)*.(CSE)*
Reg. exp. 193: PRE.(CP)*
Reg. exp. 194: (ES.(DCE)*)*.(CSE)*.PRE.(PRE.(PRE)*)*
Reg. exp. 195: PRE.(PRE.ES.(PRE.(PRE)*)*)*
Reg. exp. 196: PRE.PRE.(CP)*
Reg. exp. 197: DCE.(ES.ES.((CM)*) + (CP.(ES)*).CSE.(CSE)*)*
Reg. exp. 198: DCE.(CF.(CM)*.DCE.((DCE.(PRE.((PRE) + (DCE.(CP.(CM)*))*)*)*)*) + (PRE).(DCE)*)*
Reg. exp. 199: (((PRE.(CSE)*) + (PRE.(PRE)*).CF.(ES)*.(DCE.(DCE)*)*)
         + (CF.(ES)*).(CSE)*) + (CSE).CSE.CF.(PRE)*
Reg. exp. 200: ES.CF.(ES)*
Reg. exp. 201: CM.((PRE) + ((CF)*))*.ES.CSE.(CM)*.(CF)*
Reg. exp. 202: (CP)*.(ES)*
Reg. exp. 203: CSE.(PRE)*
Reg. exp. 204: CP.CSE.(CF)*.(DCE)*
Reg. exp. 205: ES.(PRE)*
Reg. exp. 206: (((CSE) + (DCE))*) + (CM.((CSE) + (CP))*).ES.(CSE)*
Reg. exp. 207: CM.DCE.DCE.(CF.CF.(ES)*) + ((CM)*.CP.((DCE)
         + (CF).((ES)*.PRE.(PRE)*)*)*).CF.((DCE) + (CM))*
Reg. exp. 208: ((ES.(ES)*) + (CP).(CM.PRE.(CF)*)*)*.(PRE)*
Reg. exp. 209: CSE.(CF)*
Reg. exp. 210: CM.ES.(ES)*
Reg. exp. 211: DCE.CM.CP.(((CM) + ((CF)*.(CP)*)) + (CSE.((DCE) + ((CSE)*).(PRE)*)*))*
Reg. exp. 212: CP.(PRE)*
Reg. exp. 213: CF.(CSE.ES.(((CM)*) + (DCE))*)*
Reg. exp. 214: PRE.CP.DCE.(CF.PRE.CSE.(DCE.((CSE) + (CM))*)*)*
Reg. exp. 215: CP.((CF) + (DCE))*.(ES.(CSE.(ES)*)*)*
Reg. exp. 216: CP.((CSE)*.(PRE.((CM)*.(CSE)*)
         + (CF.(PRE)*).CM.ES.(CP.(DCE.PRE.(CP)*)*)*)*)*.(ES.CF.PRE)*
Reg. exp. 217: CP.DCE.(CSE)*.(PRE)*
Reg. exp. 218: CSE.ES.(DCE)*
Reg. exp. 219: CF.((CSE.(DCE.((PRE)*.CM.(ES)*)*) + ((CP)*).(PRE)
         + (CM).CF.(CSE)*)*.CM.((PRE)*.(CP)*.(CSE)*)*)*
Reg. exp. 220: (ES.(ES.(CP)*.(CP)*)*)*.(CM.(DCE)*)*
Reg. exp. 221: DCE.(((CF) + (CF).(DCE.(CSE.(CP)*.DCE.ES.CF.(ES)*)*)*)*.(CM)*)*
Reg. exp. 222: PRE.(ES)*
Reg. exp. 223: (CP) + (CF).(CSE)*
Reg. exp. 224: CSE.(ES.((DCE)*) + (CM).(CM)*)*.(CP)*
Reg. exp. 225: CP.(CF)*
Reg. exp. 226: CM.(DCE)*
Reg. exp. 227: PRE.(PRE)*.(((CF) + (CSE.(ES)*.CSE.DCE.DCE.CM.(DCE)*))*.((PRE) + (CF))*)
         + (CM).((CSE) + (ES))*
Reg. exp. 228: ES.(DCE)*
Reg. exp. 229: CP.(CF)*
Reg. exp. 230: CP.(PRE)*
Reg. exp. 231: DCE.CF.(DCE)*.(DCE)*
Reg. exp. 232: (CM)*.((CP) + (ES))*
Reg. exp. 233: CM.(CSE)*
Reg. exp. 234: CSE.CF.CF.((((CM) + (CP)) + (PRE)) + (ES)) + (DCE).PRE.ES.(CP)*
Reg. exp. 235: DCE.CF.CM.((CP) + ((ES)*))*.ES.(CSE)*
Reg. exp. 236: PRE.(DCE)*.((((CSE) + (CP.CF.(CF)*))*) + ((PRE) + ((PRE.((CSE.(CM)*)
         + (CF))*)*).(ES)*))*
Reg. exp. 237: (DCE)*.ES.(CSE)*
Reg. exp. 238: (CM)*.(CP)*
Reg. exp. 239: CM.CSE.(CSE)*
Reg. exp. 240: DCE.(CP)*
Reg. exp. 241: ES.(CF)*
Reg. exp. 242: ES.(CM.(ES)*)*.(DCE)*
Reg. exp. 243: PRE.CSE.(((CF)*.(CP)*) + (DCE.((ES)*.(((CM.(CSE.(CM)*)*.(CSE)*)*)
         + (ES))*)*.(CSE)*).(CM)*)*
Reg. exp. 244: (PRE.(DCE)*)*.CP.(CM.PRE.(CP.(((CSE.(DCE)*)*) + ((CF)
         + ((ES)*.((DCE)*.(CF)*)*).((CF)*.DCE.CP.ES)*))*)*)*
```

```
Reg. exp. 245: CM.(CSE.(CM.(CF)*)*)*.(DCE.(PRE)*)*
Reg. exp. 246: (CSE)*.PRE.ES.CSE.DCE.(CP)*
Reg. exp. 247: CSE.((ES)*.(CSE)*)*
Reg. exp. 248: DCE.CM.CF.CM.CSE.(CP.ES.(DCE)*)*
Reg. exp. 249: PRE.(CSE)*
Reg. exp. 250: (PRE)*.(CP)*
Reg. exp. 251: (ES)*.DCE.CF.(CSE)*
Reg. exp. 252: PRE.(CF.(CM)*)*
Reg. exp. 253: CM.PRE.(ES.(((CF)*) + ((DCE)*))*)*
Reg. exp. 254: ((DCE)*.CP.(DCE)*) + (DCE.(ES)*).PRE.(CSE)*.PRE.(CP)*
Reg. exp. 255: (CSE)*.(CF)*.(CM)*
Reg. exp. 256: (CSE) + (PRE).DCE.(CP.CF.CSE.(CM.(PRE)*)*)*
Reg. exp. 257: PRE.(PRE.(CF)*)*
Reg. exp. 258: CM.(ES)*
Reg. exp. 259: CSE.(DCE)*
Reg. exp. 260: ((CP) + ((CP) + ((PRE)*)))*.(DCE)*
Reg. exp. 261: CM.((ES) + (((CF) + ((CM) + (CM)))*))*
Reg. exp. 262: DCE.CF.ES.(PRE)*
Reg. exp. 263: (CSE.(CP)*)*.(PRE)*
Reg. exp. 264: PRE.(CSE.(ES.(CF)*)*)*.((ES) + (CM))*.(DCE)*
Reg. exp. 265: CP.(CF)*
Reg. exp. 266: DCE.(CSE)*
Reg. exp. 267: PRE.CM.(CSE.(ES)*)*
Reg. exp. 268: CF.(CP)*
Reg. exp. 269: DCE.(ES)*
Reg. exp. 270: (DCE)*.PRE.CF.((CF) + (CP.ES.(DCE)*).(((CM)*) + (ES))*)*
Reg. exp. 271: CF.((CSE.(ES)*)*.(CP)*.((CF)*) + ((CP.CM.(CF.(CF)*)*)*).((PRE) + (DCE).(CF)*)*)*
Reg. exp. 272: (PRE)*.(DCE) + (CSE).(CP)*
Reg. exp. 273: (CSE) + (CM).((CSE) + (DCE))*
Reg. exp. 274: PRE.(((((CSE) + (ES))*) + (CF.CM.CM.CM.(CP)*)) + (CF).(CSE)*)*
Reg. exp. 275: ES.(DCE)*
Reg. exp. 276: PRE.(((PRE)*.(ES)*) + ((CP.(DCE.(PRE)*)*)*))*
Reg. exp. 277: CSE.(ES)*
Reg. exp. 278: CM.(CF)*
Reg. exp. 279: CSE.(PRE.(CP)*)*
Reg. exp. 280: DCE.(ES)*
Reg. exp. 281: CF.DCE.(ES.(CF)*)*.CP.((((((CSE) + (DCE))*) + (CSE))
        + (CSE))*.PRE.DCE.(CM.CM.((CF) + (ES))*)*)*
Reg. exp. 282: (CF)*.(ES.CP.(DCE.CP.ES.(DCE.(CM)*)*)*)*
Reg. exp. 283: CSE.(PRE)*
Reg. exp. 284: (CP.(CSE)*)*.DCE.(CP.(CM) + (PRE.(ES)*).(ES) + (PRE).(CM.((CF) + (ES))*)*)*
Reg. exp. 285: CSE.CSE.((CM)*) + (CP.PRE.ES.(DCE)*).(CF)*
Reg. exp. 286: CP.(ES)*
Reg. exp. 287: (CM.(CP)*)*.((PRE) + ((CSE)*))*
Reg. exp. 288: DCE.DCE.CF.CSE.(CF.(PRE)*)*
Reg. exp. 289: CP.CSE.(CM)*
Reg. exp. 290: CSE.CM.((DCE) + (((ES)*) + (CP)).(CSE)*)*
Reg. exp. 291: CP.(CF)*.(CM)*
Reg. exp. 292: CF.((ES) + ((CP)*))*
Reg. exp. 293: ES.(PRE)*
Reg. exp. 294: DCE.(CSE.(CSE) + ((DCE)*).(DCE)*)*
Reg. exp. 295: CF.(((PRE)*) + (ES).CP.(CM)*.(CM)*) + (((DCE.(CF.CF.(CP)*)*)*) + (DCE)).(CM)*
Reg. exp. 296: ES.PRE.(PRE)*
Reg. exp. 297: CSE.DCE.CM.PRE.((CP)*.CF.(CF)*)*
Reg. exp. 298: (DCE)*.(((CM) + (CP)) + (CSE))*
Reg. exp. 299: DCE.ES.(CM)*
Reg. exp. 300: (DCE)*.(CSE)*
Reg. exp. 301: CF.(PRE)*
Reg. exp. 302: ES.(DCE)*
Reg. exp. 303: ((CF.(ES.(CM.(CM)*.(ES.(((CP)*)
        + (CF.(PRE.(CF)*)*)*.(CF)*)*)*)*) + (ES).(CP)*
Reg. exp. 304: CM.PRE.((CSE) + (CSE))*
Reg. exp. 305: CP.(DCE)*
Reg. exp. 306: PRE.(ES)*
Reg. exp. 307: (DCE) + ((CF)*.ES.(CSE)
        + ((CF.((CP)*.(CSE)*.DCE.(CM)*)*)*).(CF)*.CSE.(PRE)*).(PRE.(CM.((PRE) + (CF))*)*)*
Reg. exp. 308: DCE.(ES)*
Reg. exp. 309: (CP.(CF)*)*.CM.(CM) + ((CF.(ES)*)*).CF.(DCE)*
Reg. exp. 310: ES.((CSE) + ((PRE)*))*
Reg. exp. 311: (CSE.(DCE)*) + ((((CSE)*.(CP.((((CSE)*.(CF)*)*.(ES)*)
        + (ES))*.(DCE)*)*) + (PRE))*).(PRE)*
Reg. exp. 312: CM.(PRE.CM.(ES)*)*
Reg. exp. 313: CP.CSE.(CP)*
Reg. exp. 314: ES.(ES)*
Reg. exp. 315: CF.(DCE)*
Reg. exp. 316: DCE.ES.CP.PRE.(PRE)*
Reg. exp. 317: CF.CSE.(CF)*
Reg. exp. 318: (DCE)*.((CM) + (CP))*
Reg. exp. 319: (CP)*.(PRE.(ES)*)*.(DCE)*
Reg. exp. 320: CM.(CP)*
Reg. exp. 321: DCE.CM.(CF.(ES)*)*
Reg. exp. 322: DCE.(ES)*
Reg. exp. 323: CF.CSE.((CF.((CP.(CP.(DCE)*)*) + ((CM)*))*)*) + (PRE).(CSE)*
Reg. exp. 324: CF.(CM.(CSE.CF.(CF)*) + (ES).CP.(DCE)*)*
Reg. exp. 325: (CSE)*.PRE.PRE.(CSE)*
Reg. exp. 326: (ES.(CP)*.CM.(DCE)*)*.CSE.(((CF.CM.PRE.(CF)*)*.(ES)*)*.(DCE)*)
        + (CP).(ES) + (CP).(CP)*
```

```
Reg. exp. 327: CSE.CF.(((ES)*) + (CSE).(DCE)*)*
Reg. exp. 328: PRE.(PRE)*
Reg. exp. 329: (CF) + (CSE).(PRE)*
Reg. exp. 330: DCE.(CM.(PRE)*)*
Reg. exp. 331: CM.CF.(DCE)*
Reg. exp. 332: ES.(CM)*
Reg. exp. 333: PRE.(PRE.(CP)*)*
Reg. exp. 334: ES.(CF.(CM)*)*
Reg. exp. 335: ((CSE) + (CP)) + (DCE).(CM.(CP)*)*
Reg. exp. 336: (PRE)*.(CSE)*.((CSE)*.(CM)*)*
Reg. exp. 337: ((ES)*) + ((CF)*).PRE.(DCE)*
Reg. exp. 338: CM.ES.CF.(CP.CSE.(CP)*)*
Reg. exp. 339: (CSE)*.(DCE)*
Reg. exp. 340: DCE.(PRE)*
Reg. exp. 341: CSE.(PRE)*
Reg. exp. 342: ES.CF.PRE.(CF)*
Reg. exp. 343: ((CSE)*.PRE.(ES)*)*.(CM)*
Reg. exp. 344: (CP) + ((CM.(CSE)*.((DCE) + ((DCE.(CP)*)*))*)*.PRE.CF.CM.(DCE)*).(ES)*
Reg. exp. 345: CSE.(CF)*
Reg. exp. 346: CSE.(CM)*
Reg. exp. 347: (CP) + ((PRE) + (DCE)).(CSE.(ES)*) + ((((CP.(ES)*)*.CF.(CSE)*)
        + ((PRE) + (CM).((CP)*.(CP)*)*))*).(DCE)*
Reg. exp. 348: PRE.(CF)*.ES.(CF)*
Reg. exp. 349: CSE.(PRE)*
Reg. exp. 350: ((ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 351: (CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.CF.PRE.(ES.PRE)*.(DCE.PRE.CM.ES.PRE)*
Reg. exp. 352: ((ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 353: ((CF.PRE)*.(CSE.CP.PRE)*.DCE.PRE.CM.CF.PRE.(ES.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 354: (ES.PRE)*.(CSE.CP.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE
Reg. exp. 355: ((ES.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 356: ((ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.(CSE.CP.PRE)*)*
Reg. exp. 357: (ES.PRE)*.CSE.CP.PRE.DCE.PRE.CM.ES.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE
Reg. exp. 358: (ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.(CP.CSE.PRE)*
Reg. exp. 359: ((ES.PRE)*.CSE.CP.PRE.DCE.PRE.CM.ES.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 360: (ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CSE.CP.PRE
Reg. exp. 361: ((ES.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 362: ((ES.PRE)*.(CP.CSE.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 363: (ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.(CP.CSE.PRE)*
Reg. exp. 364: (ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CP.CSE.PRE
Reg. exp. 365: ((ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 366: ((CP.CSE.PRE)*.(CF.PRE)*.(ES.PRE)*)*
Reg. exp. 367: ((ES.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 368: (ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CP.CSE.PRE
Reg. exp. 369: ((ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 370: ((ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 371: ((ES.PRE)*.CSE.CP.PRE.(DCE.PRE.CM.ES.PRE)*.(CF.PRE)*.(DCE.PRE.CM.CF.PRE)*)*
Reg. exp. 372: ((ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 373: ((ES.PRE)*.(DCE.PRE.CM.ES.PRE)*.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 374: ((ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 375: ((ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 376: ((ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.(CP.CSE.PRE)*)*
Reg. exp. 377: (ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CP.CSE.PRE
Reg. exp. 378: ((ES.PRE)*.(CSE.CP.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 379: ((ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 380: (ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.(CSE.CP.PRE)*
Reg. exp. 381: (ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CSE.CP.PRE
Reg. exp. 382: ((ES.PRE)*.(DCE.PRE.CM.ES.PRE)*.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 383: (CP.CSE.PRE.(CF.PRE)*.(ES.PRE)*)*
Reg. exp. 384: (DCE.PRE.CM.ES.PRE.(ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 385: (CP.CSE.PRE.(CF.PRE)*.(ES.PRE)*)*
Reg. exp. 386: ((DCE.PRE.CM.ES.PRE)*.(ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.(CP.CSE.PRE)*)*
Reg. exp. 387: ((ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 388: ((ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 389: (ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CP.CSE.PRE
Reg. exp. 390: (ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CP.CSE.PRE
Reg. exp. 391: (DCE.PRE.CM.ES.PRE.(ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 392: ((ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 393: ((ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.(CSE.CP.PRE)*)*
Reg. exp. 394: ((ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 395: ((ES.PRE)*.(DCE.PRE.CM.ES.PRE)*.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 396: (CSE.CP.PRE.(CF.PRE)*.(ES.PRE)*)*
Reg. exp. 397: ((ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 398: (ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CP.CSE.PRE
Reg. exp. 399: ((ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 400: ((ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.(CP.CSE.PRE)*)*
Reg. exp. 401: ((ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 402: (ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CP.CSE.PRE
Reg. exp. 403: ((ES.PRE)*.(DCE.PRE.CM.ES.PRE)*.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 404: ((ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 405: (DCE.PRE.CM.ES.PRE.(ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 406: (DCE.PRE.CM.ES.PRE.(ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 407: (DCE.PRE.CM.ES.PRE.(ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 408: ((ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 409: ((ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 410: (DCE.PRE.CM.ES.PRE.(ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 411: (CSE.CP.PRE.(CF.PRE)*.(ES.PRE)*)*
Reg. exp. 412: (DCE.PRE.CM.ES.PRE.(ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.(CSE.CP.PRE)*)*
```

```
Reg. exp. 413: (ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CSE.CP.PRE
Reg. exp. 414: ((ES.PRE)*.(DCE.PRE.CM.ES.PRE)*.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 415: ((ES.PRE)*.(DCE.PRE.CM.ES.PRE)*.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 416: (ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CSE.CP.PRE
Reg. exp. 417: ((ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 418: (CP.CSE.PRE.(CF.PRE)*.(ES.PRE)*)*
Reg. exp. 419: ((ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 420: ((ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 421: (CP.CSE.PRE.(CF.PRE)*.(ES.PRE)*)*
Reg. exp. 422: (DCE.PRE.CM.ES.PRE.(ES.PRE)*.DCE.PRE.CM.CF.PRE.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 423: (ES.PRE)*.DCE.PRE.CM.ES.PRE.(CF.PRE)*.CP.CSE.PRE
Reg. exp. 424: ((ES.PRE)*.(DCE.PRE.CM.CF.PRE)*.(CF.PRE)*.(CSE.CP.PRE)*)*
Reg. exp. 425: ((ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.CF.PRE)*
Reg. exp. 426: ((ES.PRE)*.(CF.PRE)*.(CSE.CP.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 427: ((CF.PRE)*.(ES.PRE)*.CSE.CP.PRE)*
Reg. exp. 428: ((ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.ES.PRE)*
Reg. exp. 429: (CF.PRE)*.(ES.PRE)*.(CSE.CP.PRE)*
Reg. exp. 430: ((ES.PRE)*.(CF.PRE)*.(CP.CSE.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 431: ((ES.PRE)*.(CF.PRE)*.(CSE.CP.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 432: ((ES.PRE)*.(CF.PRE)*.(CP.CSE.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 433: (ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE
Reg. exp. 434: (ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.ES.PRE
Reg. exp. 435: ((ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE)*
Reg. exp. 436: (ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.ES.PRE
Reg. exp. 437: ((CSE.CP.PRE)*.(ES.PRE)*.(CF.PRE)*.DCE.PRE.CM.CF.PRE.DCE.PRE.CM.ES.PRE)*
Reg. exp. 438: ((CP.CSE.PRE)*.(ES.PRE)*.(CF.PRE)*.(DCE.PRE.CM.CF.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 439: ((CF.PRE)*.(ES.PRE)*.CP.CSE.PRE)*
Reg. exp. 440: ((ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE)*
Reg. exp. 441: (ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.CF.PRE
Reg. exp. 442: (ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE
Reg. exp. 443: ((CF.PRE)*.(ES.PRE)*.CP.CSE.PRE)*
Reg. exp. 444: (CP.CSE.PRE.(ES.PRE)*.(CF.PRE)*.DCE.PRE.CM.CF.PRE.(DCE.PRE.CM.ES.PRE)*)*
Reg. exp. 445: ((ES.PRE)*.CSE.CP.PRE.(CF.PRE)*.(DCE.PRE.CM.CF.PRE)*)*
Reg. exp. 446: ((ES.PRE)*.(CF.PRE)*.CP.CSE.PRE)*
Reg. exp. 447: (ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE
Reg. exp. 448: ((ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 449: ((ES.PRE)*.(CP.CSE.PRE)*.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 450: ((ES.PRE)*.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 451: ((DCE.PRE.CM.CF.PRE)*.(ES.PRE)*.CSE.CP.PRE.(CF.PRE)*.(DCE.PRE.CM.ES.PRE)*)*
Reg. exp. 452: (DCE.PRE.CM.CF.PRE.(ES.PRE)*.CSE.CP.PRE.(CF.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 453: (ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE
Reg. exp. 454: (ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE
Reg. exp. 455: ((ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 456: ((DCE.PRE.CM.CF.PRE)*.(ES.PRE)*.(CSE.CP.PRE)*.(CF.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 457: (ES.PRE)*.(CP.CSE.PRE)*.(CF.PRE)*.DCE.PRE.CM.ES.PRE
Reg. exp. 458: ((ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 459: (ES.PRE)*.CSE.CP.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE
Reg. exp. 460: (DCE.PRE.CM.CF.PRE.(ES.PRE)*.(CSE.CP.PRE)*.(CF.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 461: (DCE.PRE.CM.CF.PRE.(ES.PRE)*.(CSE.CP.PRE)*.(CF.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 462: ((ES.PRE)*.(CF.PRE)*.CSE.CP.PRE)*
Reg. exp. 463: ((ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 464: ((ES.PRE)*.(CP.CSE.PRE)*.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 465: (DCE.PRE.CM.CF.PRE.(ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 466: ((ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 467: (DCE.PRE.CM.CF.PRE.(ES.PRE)*.CSE.CP.PRE.(CF.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 468: ((ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 469: (ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.ES.PRE
Reg. exp. 470: ((ES.PRE)*.(CF.PRE)*.(CSE.CP.PRE)*)*
Reg. exp. 471: (ES.PRE)*.CSE.CP.PRE.(CF.PRE)*.DCE.PRE.CM.ES.PRE
Reg. exp. 472: DCE.PRE.CM.CF.PRE.(ES.PRE)*.CP.CSE.PRE.(CF.PRE)*.DCE.PRE.CM.ES.PRE
Reg. exp. 473: ((CF.PRE)*.(ES.PRE)*.CP.CSE.PRE)*
Reg. exp. 474: ((ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE)*
Reg. exp. 475: ((ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.ES.PRE)*
Reg. exp. 476: (ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.CF.PRE
Reg. exp. 477: ((ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.CF.PRE)*
Reg. exp. 478: (ES.PRE)*.(CF.PRE)*.(CSE.CP.PRE)*.DCE.PRE.CM.ES.PRE
Reg. exp. 479: (DCE.PRE.CM.CF.PRE.(ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE)*
Reg. exp. 480: ((ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.CF.PRE)*
Reg. exp. 481: ((ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.CF.PRE)*
Reg. exp. 482: ((ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.CF.PRE)*
Reg. exp. 483: ((DCE.PRE.CM.CF.PRE)*.(ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE)*
Reg. exp. 484: ((CF.PRE)*.(ES.PRE)*.CP.CSE.PRE)*
Reg. exp. 485: (ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE
Reg. exp. 486: ((CF.PRE)*.(ES.PRE)*.CSE.CP.PRE)*
Reg. exp. 487: (ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.ES.PRE
Reg. exp. 488: ((CF.PRE)*.(ES.PRE)*.(CSE.CP.PRE)*)*
Reg. exp. 489: ((ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.(DCE.PRE.CM.CF.PRE)*)*
Reg. exp. 490: ((CF.PRE)*.(ES.PRE)*.(CP.CSE.PRE)*)*
Reg. exp. 491: ((ES.PRE)*.(CF.PRE)*.CP.CSE.PRE.DCE.PRE.CM.CF.PRE)*
Reg. exp. 492: (CF.PRE)*.(ES.PRE)*.CSE.CP.PRE
Reg. exp. 493: DCE.PRE.CM.CF.PRE.(ES.PRE)*.(CF.PRE)*.(CP.CSE.PRE)*.(DCE.PRE.CM.ES.PRE)*
Reg. exp. 494: ((ES.PRE)*.(CF.PRE)*.(CP.CSE.PRE)*.DCE.PRE.CM.CF.PRE)*
Reg. exp. 495: ((ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.CF.PRE)*
Reg. exp. 496: ((ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.ES.PRE)*
Reg. exp. 497: ((ES.PRE)*.(CF.PRE)*.(CSE.CP.PRE)*.DCE.PRE.CM.ES.PRE)*
Reg. exp. 498: (ES.PRE)*.(CF.PRE)*.CSE.CP.PRE.DCE.PRE.CM.ES.PRE
Reg. exp. 499: ((CF.PRE)*.(ES.PRE)*.CP.CSE.PRE)*
```

A.2 Table of regular expressions with the number of instructions executed

	Regular expressions with the best number of executed instructions		
	Number of instructions exec.	Number of regular expressions	Percentage
BenchPrg1	896	115	23.00%
BenchPrg2	3708603	167	33.40%
BenchPrg3	171925	151	30.20%
BenchPrg4	121216	168	33.60%
BenchPrg5	97247	500	100.00%
BenchPrg6	51710	168	33.60%
BenchPrg7	431514	500	100.00%
BenchPrg8	1590652	158	31.60%
BenchPrg9	3508	128	25.60%
BenchPrg10	3982611	168	33.60%
BenchPrg11	26407	121	24.20%
BenchPrg12	1705	168	33.60%
BenchPrg13	1108	125	25.00%
BenchPrg14	258	158	31.60%
BenchPrg15	498190	133	26.60%
BenchPrg16	14109	126	25.20%
BenchPrg17	34096	134	26.80%
		Average	37.51%

A.3 Data from metric-based phase-ordering evaluation

In this section are shown the tables for the evaluation of the metric-based phase-ordering in Section 6.1.

Without using analysis results from metrics' computation:

	Number of instructions exec.	Transformations used	Time spent on metrics in ms	Time spent in ms (overall)
BenchPrg1	896	4	214	443
BenchPrg2	3708603	2	753	1598
BenchPrg3	171925	6	368	969
BenchPrg4	121216	3	86	199
BenchPrg5	97247	0	14	14
BenchPrg6	51710	3	67	195
BenchPrg7	431514	0	11	12
BenchPrg8	1590652	4	199	387
BenchPrg9	3508	6	33	69
BenchPrg10	3982611	3	4105	14225
BenchPrg11	26407	7	48	127
BenchPrg12	1705	2	6	10
BenchPrg13	1108	6	11	23
BenchPrg14	258	4	18	44
BenchPrg15	498190	6	120	415
BenchPrg16	14109	10	31	102
BenchPrg17	34096	8	90	355

Using analysis results from metrics' computation:

	Without reuse of analysis' results		With reuse of analysis' results	
	Time spent on metrics in ms	Time spent in ms (overall)	Time spent on metrics in ms	Time spent in ms (overall)
BenchPrg1	214	443	198	251
BenchPrg2	753	1598	750	765
BenchPrg3	368	969	364	462
BenchPrg4	86	199	66	80
BenchPrg5	14	14	13	13
BenchPrg6	67	195	64	75
BenchPrg7	11	12	11	12
BenchPrg8	199	387	200	209
BenchPrg9	33	69	31	41
BenchPrg10	4105	14225	4074	4111
BenchPrg11	48	127	77	90
BenchPrg12	6	10	5	7
BenchPrg13	11	23	14	14
BenchPrg14	18	44	19	23
BenchPrg15	120	415	134	152
BenchPrg16	31	102	33	48
BenchPrg17	90	355	103	147

A.4 Data for metrics update using dependencies

Below is the table of the results for the metric-based optimization using dependencies, without reuse of metrics' analysis results for the transformation calls.

The number of transformations between parenthesis represents the number of Precalculation used; this number is counted in the total number of transformations used, while it was not when comparing the metric-based phase-ordering without using the dependencies.

	Nb. of instructions	Transformations used	Time spent in ms (metrics)	Time spent in ms (overall)
BenchPrg1	896	4 (0)	198	434
BenchPrg2	3708603	3 (1)	986	1868
BenchPrg3	171925	10 (4)	541	1175
BenchPrg4	121216	4 (1)	142	265
BenchPrg5	97247	0 (0)	12	13
BenchPrg6	51710	4 (1)	103	239
BenchPrg7	431514	0 (0)	8	8
BenchPrg8	1590652	5 (1)	281	469
BenchPrg9	3508	11 (3)	41	127
BenchPrg10	3982611	4 (1)	5267	15427
BenchPrg11	26407	12 (3)	74	167
BenchPrg12	1705	3 (1)	8	13
BenchPrg13	1108	7 (1)	13	31
BenchPrg14	258	5 (1)	20	54
BenchPrg15	498190	9 (2)	210	503
BenchPrg16	14109	15 (3)	60	152
BenchPrg17	34096	12 (4)	137	408

The other table below represents the results for the metric-based optimization using dependencies with reuse of metrics' analysis results.

	Nb. of instructions	Transformations used	Time spent in ms (metrics)	Time spent in ms (overall)
BenchPrg1	896	4	216	256
BenchPrg2	3708603	3	974	998
BenchPrg3	171925	10	509	601
BenchPrg4	121216	4	137	160
BenchPrg5	97247	0	11	11
BenchPrg6	51710	4	103	113
BenchPrg7	431514	0	8	8
BenchPrg8	1590652	5	283	296
BenchPrg9	3508	11	63	78
BenchPrg10.	3982611	4 (1)	5274	5318
BenchPrg11	26407	12 (3)	80	98
BenchPrg12	1705	3 (1)	7	8
BenchPrg13	1108	7 (1)	19	19
BenchPrg14	258	5 (1)	21	27
BenchPrg15	498190	9 (2)	179	227
BenchPrg16	14109	15 (3)	66	91
BenchPrg17	34096	12 (4)	147	190

A.5 Table of data for size-aimed optimization

	Size of the program	Transformations used	Time spent in ms (overall)
BenchPrg1	365	4	251
BenchPrg2	1781	2	765
BenchPrg3	698	6	462
BenchPrg4	431	3	80
BenchPrg5	536	0	13
BenchPrg6	438	3	75
BenchPrg7	582	0	12
BenchPrg8	411	4	209
BenchPrg9	87	6	41
BenchPrg10	3731	3	4111
BenchPrg11	192	7	90
BenchPrg12	89	2	7
BenchPrg13	111	6	14
BenchPrg14	62	4	23
BenchPrg15	568	6	152
BenchPrg16	185	10	48
BenchPrg17	402	8	147

Above is the table of data for metric-based phase-ordering without weights. Below is the table of data for metric-based phase-ordering using weights and detection of non-improving transformation calls:

	Size of the program	Transformations used	Time spent in ms (overall)
BenchPrg1	331	4	228
BenchPrg2	1781	2	818
BenchPrg3	698	7	566
BenchPrg4	431	4	118
BenchPrg5	536	0	15
BenchPrg6	438	4	133
BenchPrg7	582	0	8
BenchPrg8	411	4	199
BenchPrg9	87	6	26
BenchPrg10	3731	3	4437
BenchPrg11	192	9	72
BenchPrg12	89	2	8
BenchPrg13	111	6	14
BenchPrg14	62	4	24
BenchPrg15	568	6	219
BenchPrg16	185	12	59
BenchPrg17	402	10	152

Appendix B

Comparison between the different algorithms

This appendix contains the different data dealing with the comparison between the different algorithms available (MFP algorithm, Abstract Worklist algorithm and Propagation algorithm). The first section compares the outputs of these algorithms, while the second section contains the data for the comparison of performance for the Copy Analysis performed in Section 5.3.6.

B.1 Equality of the results

In order to evaluate the efficiency of the Propagation algorithm, its results have been compared to the ones from two classical algorithms (the MFP algorithm and the Abstract Worklist algorithm). A Java class *TestAlgorithms* (in the package *optimizer.manager.util*) has been implemented to evaluate if the Propagation algorithm was giving the same results as the MFP algorithm for the different data-flow analyses, as the MFP algorithm and the Abstract Worklist Algorithm are already known to give the same results.

In this test class has been implemented several test methods:

1. Method *testCopyAnalysis()*: this method tests the equality between the results of the two algorithms (MFP and Propagation) for the Copy Analysis.

2. Method *testReachingDefinitions()*: this method tests the equality between the results of the two algorithms (MFP and Propagation) for the Reaching Definitions Analysis.

3. Method *testAvailableExpressions()*: this method tests the equality between the results of the two algorithms (MFP and Propagation) for the Available Expressions Analysis.

4. Method *testLiveVariables()*: this method tests the equality between the results of the two algorithms (MFP and Propagation) for the Live Variables Analysis.

5. Method *testVeryBusyExpressions()*: this method tests the equality between the results of the two algorithms (MFP and Propagation) for the Very Busy Expressions Analysis.

6. Method *testConstantPropagation()*: this method tests the equality between the results of the two algorithms (MFP and Propagation) for the Constant Propagation Analysis.

7. Method *testDetectionOfSigns()*: this method tests the equality between the results of the two algorithms (MFP and Propagation) for the Detection of Signs Analysis.

8. Method *testAllPrograms()*: this method uses all the previous method to test all the analyses available on all benchmark programs.

Each of these methods gets the results from the Propagation algorithm, then transforms the shape of these results so it can fit the same layout than the results from the MFP algorithm, and finally tests the equality of the resulting mappings. This transformation is necessary as the Propagation algorithm does not use exactly the same lattices than the one in the MFP algorithm, though it deals with the same information. Thus, adapting the results' organization is important in order to be able to directly test for equality.

The following table shows the results of these tests:

Bench Program	RD	LV	CA	AE	VBE	CP	DOS
1	√	√	√	√	√	√	√
2	√	√	√	√	√	√	√
3	√	√	√	√	√	√	√
4	√	√	√	√	√	√	√
5	√	√	√	√	√	√	√
6	√	√	√	√	√	√	√
7	√	√	√	√	√	√	√
8	√	√	√	√	√	√	√
9	√	√	√	√	√	√	√
10	√	√	√	√	√	√	√
11	√	√	√	√	√	√	√
12	√	√	√	√	√	√	√
13	√	√	√	√	√	√	√
14	√	√	√	√	√	√	√
15	√	√	√	√	√	√	√
16	√	√	√	√	√	√	√
17	√	√	√	√	√	√	√

On this table, RD stands for "Reaching Definitions Analysis", LV for "Live Variables Analysis", CA for "Copy Analysis", AE for "Available Expressions Analysis", VBE for "Very Busy Expressions Analysis", CP for "Constant Propagation Analysis", and DOS for "Detection of Signs Analysis".

Performing these tests ensured that the two algorithms computed the same results on all the benchmark programs, which permits to declare that the Propagation algorithm gives exactly the same results as the classical algorithms.

B.2 Comparison of performance

The following table contains the running time of the different algorithms for the calculation of Copy Analysis on the different benchmark programs.

	Running time for Copy Analysis (in ms)		
Bench Program	MFP algorithm	Abstract Worklist algorithm	Propagation algorithm
1	156	125	47
2	2891	904	188
3	372	218	94
4	110	109	47
5	77	62	49
6	94	78	46
7	16	31	16
8	78	63	31
9	47	47	30
10	4968	4562	515
11	116	84	78
12	46	63	16
13	125	62	15
14	82	63	31
15	110	78	63
16	172	125	67
17	139	109	58

As it is explained in Section 5.3.6, this comparison shows that the Propagation algorithm is much faster than the two other algorithms (MFP and Abstract Worklist algorithms), while giving exactly the same information.

Appendix C

Dependencies and effects of the different transformations

This appendix aims at completing the analyses of the transformations' dependencies and effects made in the previous chapters (Section 6.2.2 and Section 7.2).

C.1 Dependencies

C.1.1 Common Subexpression Elimination

This part deals with the connections of the Common Subexpression Elimination transformation. It describes the different effects that can have the transformation on all the other ones. The (+) symbol means that there is a connection from Common Subexpression Elimination, where (-) means there is not.

- **Dead Code Elimination (+).** Common Subexpressions Elimination is connected to the Dead Code Elimination transformation: indeed, the use of Common Subexpressions Elimination can increase the number of time the Dead Code Elimination can be applied, by introducing faint temporary

variables. Consider the following program:

$$[\texttt{x:=a+b}]^1; [\texttt{y:=a+b}]^2$$

Dead Code Elimination can remove the two assignments labeled 1 and 2. By applying the Common Subexpressions Elimination:

$$\Rightarrow_{CSE} [\texttt{tp:=a+b}]^3; [\texttt{x:=tp}]^1; [\texttt{y:=tp}]^2$$

Now the Dead Code Elimination can remove all three assignments, as the variable *tp* is faint (i.e only used in dead assignments).

- **Constant Folding (+).** Common Subexpressions Elimination is also linked to the Constant Folding transformation: the use of Common Subexpressions Elimination can reduce the number of time the Constant Folding can replace variables by their constant value, because expressions are calculated less often, as shown in the example:

$$[\texttt{b:=3}]^1; [\texttt{x:=b-a}]^2; [\texttt{y:=b-a}]^3$$

Here Constant Folding can be applied once at label 2 and once at label 3, replacing variable *b* by its value 3. After performing Common Subexpressions Elimination:

$$\Rightarrow_{CSE} [\texttt{b:=3}]^1; [\texttt{tp:=b-a}]^4; [\texttt{x:=tp}]^2; [\texttt{y:=tp}]^3$$

the Constant Folding transforming can only be applied once at label 4.

- **Copy Propagation (+).** Common Subexpressions Elimination can enable the Copy Propagation transformation when replacing available expressions by temporary variables. Consider the following example:

$$[\texttt{x:=a+b}]^1; [\texttt{y:=a+b}]^2; \texttt{write } [\texttt{x+c}]^3$$

Copy Propagation cannot be applied here, but Common Subexpressions Elimination can. Once it is applied, Copy Propagation can also be applied:

$$\begin{gathered}\Rightarrow_{CSE} [\texttt{tp:=a+b}]^4; [\texttt{x:=tp}]^1; [\texttt{y:=tp}]^2; \texttt{write } [\texttt{x+c}]^3 \\ \Rightarrow_{CP} [\texttt{tp:=a+b}]^4; \texttt{write } [\texttt{tp+tp}]^3\end{gathered}$$

- **Precalculation Process (+).** Common Subexpressions Elimination modify the Precalculation Process' results. Indeed, as it aims at reducing the number of computation of some expressions, the Precalculation Process' efficiency is reduced whenever these expressions are constant.For example in the program:

$$[\texttt{x:=3+4}]^1; [\texttt{y:=3+4}]^2$$

the Precalculation Process is applied twice (at label 1 and 2), while after Common Subexpressions Elimination:

$$\begin{array}{c} \Rightarrow_{CSE} [\texttt{tp:=3+4}]^3; [\texttt{x:=tp}]^1; [\texttt{y:=tp}]^2 \\ \Rightarrow_{PRE} [\texttt{tp:=7}]^3; [\texttt{x:=tp}]^1; [\texttt{y:=tp}]^2 \end{array}$$

here the Precalculation Process is only applied once at label 3.

- **Code Motion (+).** The last transformation, Code Motion, is influenced by Common Subexpressions Elimination as well. It simply comes from the fact that Common Subexpressions Elimination introduces new assignments: these assignments can be loop invariants as well and thus increase the number of invariants to move out of the loop by the Code Motion transformation. Consider this example program:

$$\texttt{while } [\texttt{true}]^1 \texttt{ do } [\texttt{x:=a+b}]^2; [\texttt{y:=a+b}]^3 \texttt{ od}$$

$$\begin{array}{l} \Rightarrow_{CM} \texttt{if } [\texttt{true}]^4 \texttt{ then } ([\texttt{x:=a+b}]^5; [\texttt{y:=a+b}]^6; \\ \qquad \texttt{while } [\texttt{true}]^1 \texttt{ do } [\texttt{skip}]^3 \texttt{ od}) \texttt{ fi} \end{array}$$

Code Motion can be applied to the two loop invariants labeled 2 and 3. However, if Common Subexpressions Elimination is used before, the extra assignment created is also a loop invariant and can be moved as well, so three invariants are concerned by Code Motion:

$$\Rightarrow_{CSE} \texttt{while } [\texttt{true}]^1 \texttt{ do } [\texttt{tp:=a+b}]^4; [\texttt{x:=tp}]^2; [\texttt{y:=tp}]^3 \texttt{ od}$$

$$\begin{array}{l} \Rightarrow_{CM} \texttt{if } [\texttt{true}]^5 \texttt{ then } ([\texttt{tp:=a+b}]^6; [\texttt{x:=tp}]^7; [\texttt{y:=tp}]^8; \\ \qquad \texttt{while } [\texttt{true}]^1 \texttt{ do } [\texttt{skip}]^3 \texttt{ od}) \texttt{ fi} \end{array}$$

- **Elimination with Signs (-).** The Elimination with Signs transformation's results are not modified by the use of Common Subexpressions Elimination, because Elimination with Signs deals with evaluating non-trivial boolean expressions, while Common Subexpressions Elimination just replaces non-trivial arithmetic expressions. Thus, even if the arithmetic expressions inside a boolean expression are replaced by temporary variables, the Detection of Signs analysis will still be able to figure out the sign of the boolean expression if it could do it before Common Subexpressions Elimination.

C.1.2 Copy Propagation

This part deals with the connections of the Copy Propagation transformation. The (+) symbol means that there is a connection from Copy Propagation, where (-) means there is not.

- **Common Subexpressions Elimination (+).** Copy Propagation can enable the Common Subexpressions Elimination transformation when replacing variables by their associated copy. Consider the following example:

  $$[\texttt{c:=b}]^1; [\texttt{x:=a+b}]^2; [\texttt{y:=a+c}]^3$$

  Common Subexpressions Elimination cannot be applied here, but Copy Propagation can. Once it is applied, Common Subexpressions Elimination can also be applied:

  $$\begin{aligned} &\Rightarrow_{CP} [\texttt{x:=a+b}]^2; [\texttt{y:=a+b}]^3 \\ &\Rightarrow_{CSE} [\texttt{tp:=a+b}]^4; [\texttt{x:=tp}]^2; [\texttt{y:=tp}]^3 \end{aligned}$$

- **Code Motion (+).** Code Motion is influenced as well by the Copy Propagation transformation. It comes from the fact that Copy Propagation can remove some assignments inside loops and thus allows some statements to become invariants. Consider this example program:

  $$[\texttt{x:=y}]^1; \texttt{while } [\texttt{x<3}]^2 \texttt{ do } [\texttt{x:=y}]^3; [\texttt{t:=x+2}]^4 \texttt{ od}$$

  Here Code Motion cannot be applied, as x is redefined in the loop. But it is possible to use Code Motion once Copy Propagation has been applied:

  $$\begin{aligned} &\Rightarrow_{CP} \texttt{while } [\texttt{y<3}]^2 \texttt{ do } [\texttt{t:=y+2}]^4 \texttt{ od} \\ &\Rightarrow_{CM} \texttt{if } [\texttt{y<3}]^5 \texttt{ then } ([\texttt{t:=y+2}]^6; \\ &\quad \texttt{while } [\texttt{y<3}]^2 \texttt{ do } [\texttt{skip}]^4 \texttt{ od}) \texttt{ fi} \end{aligned}$$

- **Constant Folding (+).** A straightforward example shows that Constant Folding has dependencies with Copy Propagation:

  $$[\texttt{a:=3}]^1; [\texttt{x:=a}]^2; [\texttt{y:=x*2}]^3$$

  Here Constant Folding can be applied once at label 2 and once at label 3, replacing variable a and x by their value 3. After performing Copy Propagation:

  $$\Rightarrow_{CP} [\texttt{a:=3}]^1; [\texttt{y:=a*2}]^3$$

Constant Folding can only be applied once at label 3.

- **Dead Code Elimination (+).** When replacing variables by their copy, the Copy Propagation transformation makes some copy assignments become dead. However, the very Copy Propagation deletes these assignments by himself, so at the end of the transformation there is no more dead assignment than there was before the execution of the transformation, and no less, since the dead copy assignments are not removed by this transformation. However, Copy Propagation may also remove assignments to faint variables. Consider the following example:

  $$[\texttt{x:=y}]^1; [\texttt{z:=4*x}]^2$$

  Dead Code Elimination can be applied on both assignments, as z is dead at label 2 and x is faint at label 1. However, if Copy Propagation is applied:

  $$\Rightarrow_{CP} [\texttt{z:=4*y}]^2$$

  Dead Code Elimination can only be applied once.

- **Precalculation Process (-).** The Precalculation Process' results are not modified by the use of Copy Propagation. Indeed, Copy Propagation affects only copy variables that are replaced by other variables (so that does not influence anything in the Precalculation), and can delete copy assignments, which could not be improved by the Precalculation.

- **Elimination with Signs (-).** As for the Dead Code Elimination, the Elimination with Signs transformation is not influenced by the use of Copy Propagation. This transformation aims at statically computing expressions when the signs of the operands are known, so whenever a variable is replaced by its copy variable, their are supposed to have the same value, so a fortiori the same sign. Thus, the Elimination with Signs transformation will have exactly the same effects of the variables are replaced by their copy variables. Moreover, the fact that copy assignments become dead and are deleted does not influence in any case the Elimination with Signs transformation, which deals with non-trivial expressions only (variables are trivial expressions).

C.1.3 Dead Code Elimination

This part deals with the connections of the Dead Code Elimination transformation.

- **Common Subexpressions Elimination (+).** Dead Code Elimination can modify Common Subexpressions Elimination's results when deleting dead assignments containing available expressions. Consider the following example:

$$[\texttt{x:=a+b}]^1; [\texttt{y:=a+b}]^2; \texttt{write } [|x|]^3$$

  Common Subexpressions Elimination can be applied here, because `a+b` is available at the entry of label 2. But if Dead Code Elimination is applied:

$$\Rightarrow_{DCE} [\texttt{x:=a+b}]^1; \texttt{write } [|x|]^3$$

  Now Common Subexpressions Elimination cannot be applied any more.

- **Code Motion (+).** Code Motion is influenced as well by Dead Code Elimination, in a similar way as Common Subexpressions Elimination. It simply comes from the fact that if an assignment to a dead (or faint) variable is a loop invariant, it can be moved out of the loop if Code Motion is used, but if Dead Code Elimination is used before, then Code Motion will be useless.

- **Precalculation Process (+).** Again, the same reason as for the previous transformation applies for the Precalculation Process. If a dead assignment contains a non-trivial expressions with only constant values involved, the Precalculation Process will be able to statically compute their results, while if Dead Code Elimination is used before, the Precalculation Process will be of no use.

- **Constant Folding (+).** It is pretty obvious to see that Constant Folding has dependencies with Dead Code Elimination:

$$[\texttt{a:=-1}]^1; [\texttt{x:=(a>0)}]^2$$

  Here Constant Folding can be applied once at label 2, while after Dead Code Elimination (a and x are dead variables), Constant Folding cannot be applied at all:

$$\Rightarrow_{DCE} [\texttt{skip}]^2$$

  Constant Folding can only be applied once at label 3.

- **Elimination with Signs (+).** The same example as for Constant Folding can be used to show that Dead Code Elimination can influence Elimination with Signs' results as well. In the original program, Elimination with Signs can be applied at label 2 to replace `a>0` by `false`, while it cannot be applied after the use of Dead Code Elimination.

- **Copy Propagation (+).** Finally, the Copy Propagation transformation can be, as all the others transformations, sensitive to the use of Dead Code Elimination: if the copy assignments considered are dead assignments (or faint), Dead Code Elimination will delete them, and Copy Propagation will not be able to be useful after:

$$[\texttt{x:=y}]^1; [\texttt{y:=2*x}]^2 \Rightarrow_{DCE} [\texttt{skip}]^2$$

  Here the variable x is faint, and y is dead, so everything is removed, and Copy Propagation cannot be applied afterwards.

C.1.4 Code Motion

This part deals with the connections of the Code Motion transformation. The (+) symbol means that there is a connection from Code Motion, where (-) means there is not.

- **Precalculation Process (+).** The Precalculation Process' results are modified by the use of Code Motion. When moving code outside a while loop, Code Motion creates if statements to guard the pre-header containing loop invariants, and thus the Precalculation Process may simplify the if statement if the condition is constant.
- **Constant Folding (+).** Constant Folding is influenced by Code Motion, for example because Constant Folding can be enable in the condition of the if statements, as can be seen in this example:

$$[\texttt{x:=2}]^1; \texttt{while } [\texttt{x>3}]^2 \texttt{ do } [\texttt{t:=5}]^3; [\texttt{x:=6}]^4 \texttt{ od}$$

  Here Constant Folding cannot be applied, but once Code Motion has been applied, Constant Folding can replace x in the created if statement condition:

$$\Rightarrow_{CM} [\texttt{x:=2}]^1; \texttt{if } [\texttt{x>3}]^5 \texttt{ then } ([\texttt{t:=5}]^6; \texttt{while } [\texttt{x>3}]^2 \texttt{ do } [\texttt{x:=6}]^4 \texttt{ od) fi}$$

$$\Rightarrow_{CF} [\texttt{x:=2}]^1; \texttt{if } [\texttt{2>3}]^5 \texttt{ then } ([\texttt{t:=5}]^6; \texttt{while } [\texttt{x>3}]^2 \texttt{ do } [\texttt{x:=6}]^4 \texttt{ od) fi}$$

- **Elimination with Signs (+).** The same reason as for Constant Folding applies for Elimination with Signs: as Code Motion introduces a new

condition with the if statement guarding the pre-header, Elimination with Signs may be enabled to compute the boolean expressions involved in this condition.

- **Common Subexpressions Elimination (+).** Again, as Code Motion duplicates the condition of the while loop in order to create the if statement, an available expression involved in this condition can have to be replaced by Common Subexpressions Elimination once more, hence results of this transformation are modified.

- **Copy Propagation (-).** The use of Code Motion does not modify the efficiency of the Copy Propagation transformation. Indeed, Code Motion does not introduce nor delete any copy assignments, and even if adding an if statement condition can force the Copy Propagation to replace more variables by their copy, their will still be the same number of copy assignments removed (which is the only *real* optimization in the transformation).

- **Dead Code Elimination (-).** Dead Code Elimination is not influenced at all by the use of Code Motion. Code Motion does not create any new assignment, nor delete any blocks, so no variables can become dead after Code Motion. The introduction of the if statement cannot make dead variables become live, as the only variables used in the if statement's condition are the variables already used in the while loop's condition.

C.1.5 Elimination with Signs

This part deals with the connections of the Elimination with Signs transformation. The (+) symbol means that there is a connection from Elimination with Signs, where (-) means there is not.

- **Precalculation Process (+).** The Elimination with Signs clearly influences the Precalculation Process by being able to replace if statements' and loops' conditions by their boolean value when this one can be guessed using the Detection of Signs analysis. Thus the Precalculation Process can simplify the if statements and the while loops (if the condition is false).

- **Code Motion (+).** As with Constant Folding, the Code Motion transformation is influenced by the Elimination with Signs. Consider this example:

$$\texttt{while } [\texttt{x<3}]^1 \texttt{ do } [\texttt{x:=1}]^2; [\texttt{p:=x>0}]^3 \texttt{ od}$$

Code Motion cannot be applied at label 3, because it uses x assigned in the loop, and which is not an invariant (since x is used in the condition as well). However, after Elimination with Signs, the label 3 can be moved out of the loop:

$$\begin{aligned} &\Rightarrow_{ES} \texttt{while } [\texttt{x<3}]^1 \texttt{ do } [\texttt{x:=1}]^2; [\texttt{p:=true}]^3 \texttt{ od} \\ \Rightarrow_{CM} &\texttt{if } [\texttt{x<3}]^4 \texttt{ then } ([\texttt{p:=true}]^5; \texttt{while } [\texttt{x<3}]^1 \texttt{ do } [\texttt{x:=1}]^2 \texttt{ od}) \texttt{ fi} \end{aligned}$$

- **Common Subexpressions Elimination (+).** By replacing expressions by their constant boolean value, the Elimination with Signs can obviously reduce the number of available expressions and then the efficiency of the Common Subexpressions Elimination.

- **Copy Propagation (+).** Copy Propagation is influenced by Elimination with Signs as well, because Elimination with Signs can remove the use of some variables and thus make some copy assignments become dead, where the Dead Code Elimination will have to delete them, instead of Copy Propagation. Consider the following example:

$$[\texttt{x:=1}]^1; [\texttt{y:=x}]^2; [\texttt{p:=y>-1}]^3$$

  Copy Propagation could propagate x to label 3, and eliminate the copy assignment of label 2. But if applied, Elimination with Signs will see that y is strictly positive, and thus evaluate `y>-1` at label 3 by `true`:

$$\Rightarrow_{ES} [\texttt{x:=1}]^1; [\texttt{y:=x}]^2; [\texttt{p:=true}]^3$$

  Now the copy assignment at label 2 is not used anymore, so the Copy Propagation will not delete it.

- **Dead Code Elimination (+).** As seen for the Copy Propagation, Elimination with Signs can create new dead assignments, and thus increase the efficiency of the Dead Code Elimination transformation.

- **Constant Folding (+).** The last transformation, Constant Folding, can also be enabled by the Elimination with Signs transformation, as can be seen in the following example:

$$\texttt{if } [\texttt{b}]^1 \texttt{ then } [\texttt{x:=3}]^2 \texttt{ else } [\texttt{x:=4}]^3 \texttt{ fi}; [\texttt{t:=x>0}]^4; \texttt{write } [\texttt{t}]^5$$

  Constant Folding cannot be used at any place here, especially not in label 4, because x can have the value of 3 or 4. But after Elimination with Signs:

$$\begin{aligned} &\Rightarrow_{ES} \texttt{if } [\texttt{b}]^1 \texttt{ then } [\texttt{x:=3}]^2 \texttt{ else } [\texttt{x:=4}]^3 \texttt{ fi}; [\texttt{t:=true}]^4; \texttt{write } [\texttt{t}]^5 \\ \Rightarrow_{CF} &\texttt{if } [\texttt{b}]^1 \texttt{ then } [\texttt{x:=3}]^2 \texttt{ else } [\texttt{x:=4}]^3 \texttt{ fi}; [\texttt{t:=true}]^4; \texttt{write } [\texttt{true}]^5 \end{aligned}$$

Elimination with Signs has spotted that x is strictly positive, and evaluate the expression at label 4 to `true`. Then the Constant Folding has been able to replace t at label 5 by `true`.

C.1.6 Precalculation Process

The last transformation, the Precalculation Process, is capable of removing a whole group of statements by simplifying if statements for example, when the condition is a boolean constant. Thus, every possible transformations seen before could have anything to do in this group of statements, and thus may see their efficiency reduce.

Hence the Precalculation Process can influence *every* transformations seen above.

C.2 Effects

In this section are described the effects on the size and the speed of the program of all the transformations other than Constant Folding.

C.2.1 Copy Propagation

The transformation described in this section is the Copy Propagation transformation, described in Section 3.4.3.

Functional effect:

The aim of this optimization is to look for the variables involved in copy assignments (assignments of type $[x := y]^l$). Then the optimization replaces further use of the variable assigned in a copy assignment by its copy variable, and removes the copy assignments associated.

Consequences on the program:

When replacing a variable's occurrences by its copy variable, the variable itself is less and less used. In fact, if all the occurrences of the variable between the copy assignment and the next assignment to this variable

are replaced by another variable (the copy variable), the variable becomes dead at this point of the program. The copy assignments are then removed, so the total number of copy assignments decreases.

⇒ Conclusion:

* **Size ⊕:** Copy Propagation does decrease the size of the program, as dead copy assignments are removed, except in the case the name of the copy variable is much longer than the one of the variable considered (long enough to counterbalanced the removal of the dead assignment). However, the latter situation may not happen very often.
* **Speed ⊕:** Copy Propagation will improve the speed of the program, as, aside from variables replaced by other variables, the noticeable changes to the program are the deletion of the copy assignments, which means one assignment less to compute.

C.2.2 Elimination with Signs

The transformation described in this section is the Elimination with Signs transformation, described in Section 3.4.6.

Functional effect:

The aim of this optimization is to statically evaluate boolean expressions whenever their value can be predicted from the sign of the variables involved in it.

Consequences on the program:

By replacing boolean expressions by their value, the transformation decreases the number of use of some variables. Indeed, the variables used in an expression which has a sign that can be predicted, are not used in the expression anymore after the expression has been changed to a constant value. Hence, these variables are less used, and in some cases they become dead. The assignments to these variables become then dead assignments (i.e assignments to a dead variable). However, as for Constant Folding, these dead assignments are not removed by the transformation. Thus, only the static evaluation of boolean expressions is performed.

⇒ Conclusion:

* **Size ⊘:** As Constant Folding, the way Elimination with Signs will modify the size of the program varies depending on the length of the name of the variables composing the expressions evaluated.
* **Speed ⊕:** Elimination with Signs improves the speed of the program, as runtime expressions evaluations (i.e variables evaluations) are replaced by constant boolean values.
* **With Precalculation:** Elimination with Signs improves **speed** and might also improve the **size**, as it may create if and while statements with constant conditions that will be deleted by Precalculation.

C.2.3 Dead Code Elimination

The transformation described in this section is the Dead Code Elimination transformation, described in Section 3.4.4.

Functional effect:

The aim of this optimization is simply to look for dead and faint assignments and to remove them.

Consequences on the program:

The only (but important) change to the program is the deletion of all assignments to dead (and faint) variables.
⇒ Conclusion:

* **Size ⊕:** Dead Code Elimination improves the size only by removing the dead assignments.
* **Speed ⊕:** Dead Code Elimination improves the speed of the program, as dead assignments may be deleted, which means less assignments to compute.

C.2.4 Common Subexpressions Elimination

The transformation described in this section is the Common Subexpressions Elimination transformation, described in Section 3.4.2.

Functional effect:

This transformation replaces expressions used more than once during a path of the program by temporary variables, and assigns the temporary variables to these expressions, in order to compute them less often.

Consequences on the program:

As the transformation replaces an expression used at least twice by a temporary variable, and compute it only when assigning it to the temporary variable, the expression will be computed less often. Thus the number of occurrences of available expressions in the program will decrease. On the other hand, new assignments are inserted.

⇒ Conclusion:

* **Size ⊘:** Common Subexpression Elimination may degrade the size because, however some expressions are replaced by a single variable, new assignments are created. Thus, the more expressions replaced by a temporary variables there are, the more the size will be likely to be improved.
* **Speed ⊕:** Common Subexpression Elimination improves the speed of the program, as available expressions are computed less often.

C.2.5 Code Motion

The transformation described in this section is the Code Motion transformation, described in Section 3.4.5.

Functional effect:

This transformation operates on loops, where the loop invariants are moved outside the loop, to a pre-header.

Consequences on the program:

As a direct consequence from the functional effect of the transformation, the number of loop invariants in all the loops is decreasing. And because these invariants are assignments, the total number of assignments in the

loop bodies should also decrease. However, the transformation will introduce an *if statement* as pre-header if there is any invariant to move outside a loop.
⇒ Conclusion:

* **Size ⊖:** Code Motion degrades the size because if pre-header is added if invariants are found.
* **Speed ⊕:** Code Motion improves the speed of the program, as loop invariant statements are moved outside the loops, so they are computed less often, except in the few cases where a loop's body is computed only once, which is unlikely.
* **With Precalculation:** Code Motion improves **speed**, and may degrade **size**: with Precalculation, there will not be less code than before the Code Motion transformation, as the invariants are not deleted, just moved. The only case where the size is not degraded is when the condition of the loop is `true`: in this case, the if pre-header is simplified by the Precalculation Process.

Bibliography

[1] Aho, Lam, Sethi, and Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson International Edition, second edition, 2007.

[2] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. *SIGPLAN Not.*, 39(7):231–239, 2004.

[3] L. N. Chakrapani, P. Korkmaz, V. J. Mooney III, K. V. Palem, K. Puttaswamy, and W. F. Wong. The emerging power crisis in embedded processors: What can a (poor) compiler do? In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 176 – 180, 2001.

[4] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, 1995.

[5] Wikipedia: Compiler. http://en.wikipedia.org/wiki/Compiler.

[6] J. Davidson and D. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472, November 1991.

[7] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proceedings of the 37th conference on Design automation*, pages 304 – 307, 2000.

[8] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. Fast searches for effective optimization phase

sequences. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 171–182. ACM Press, 2004.

[9] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23. ACM Press, 2003.

[10] Prasad A. Kulkarni, David B. Whalley, and Gary S. Tyson. Evaluating heuristic optimization phase order search algorithms. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 157–169. IEEE Computer Society, 2007.

[11] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. Exhaustive optimization phase order space exploration. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 306–318. IEEE Computer Society, 2006.

[12] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. In search of near-optimal optimization phase orderings. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 83–92. ACM Press, 2006.

[13] Han Lee, Daniel von Dincklage, Amer Diwan, and J. Eliot B. Moss. Understanding the behavior of compiler optimizations. *Softw. Pract. Exper.*, 36(8):835–844, 2006.

[14] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 1965.

[15] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[16] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[17] H. R. Nielson and F. Nielson. *Semantics with Applications: An Appetizer*. Springer, 2007.

[18] GCC Optimize options. http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[19] Markus Püschel, José Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. Spiral:

Code generation for dsp transforms. *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Platform Adaptation*, February 2005.

[20] Kevin Scott. On proebsting's law. http://citeseer.ist.psu.edu/446305.html.

[21] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, thumb, and the arm7tdmi. *IEEE Micro*, 15(5):22 – 30, 1995.

[22] S. Touati and D. Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd conference on Computing frontiers*, pages 147 – 156, 2006.

[23] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 204–215. IEEE Computer Society, 2003.

[24] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. *SIGPLAN Not.*, 25(3):137–146, 1990.

[25] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, 1997.

[26] Min Zhao, Bruce R. Childers, and Mary Lou Soffa. A model-based framework: An approach for profit-driven optimization. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 317–327. IEEE Computer Society, 2005.