Multi-objective Improvement of Software Using Co-evolution and Smart Seeding

Andrea Arcuri[1], David Robert White[2], John Clark[2], and Xin Yao[1]

[1] The Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA), The School of Computer Science, The University of Birmingham, Edgbaston, Birmingham, B15 2TT, UK
[2] Department of Computer Science, University of York, YO10 5DD, UK

Abstract. Optimising non-functional properties of software is an important part of the implementation process. One such property is execution time, and compilers target a reduction in execution time using a variety of optimisation techniques. Compiler optimisation is not always able to produce semantically equivalent alternatives that improve execution times, even if such alternatives are known to exist. Often, this is due to the local nature of such optimisations. In this paper we present a novel framework for optimising existing software using a hybrid of evolutionary optimisation techniques. Given as input the implementation of a program or function, we use Genetic Programming to evolve a new semantically equivalent version, optimised to reduce execution time subject to a given probability distribution of inputs. We employ a co-evolved population of test cases to encourage the preservation of the program's semantics, and exploit the original program through seeding of the population in order to focus the search. We carry out experiments to identify the important factors in maximising efficiency gains. Although in this work we have optimised execution time, other non-functional criteria could be optimised in a similar manner.

1 Introduction

Software developers must not only implement code that adheres to the customer's functional requirements, but they should also pay attention to performance details. There are many contexts in which the execution time is important, for example to aid performance in high-load server applications, or to maximise time spent in a power-saving mode in software for low-resource systems. Typical programmer mistakes may include the use of an inefficient algorithm or data structure, such as employing an $\Theta(n^2)$ sorting algorithm.

Even if the correct data structures and algorithms are employed, their actual implementations might still be improved. In general, compilers cannot restructure a program's implementation without restriction, even if employing semantics-preserving transformations. The alternative of relying on manual optimisation is not always possible: the performance implications of design decisions may be dependent on low-level details hidden from the programmer, or be subject to subtle interactions with other properties of the software.

To complicate the problem, external factors contribute to the execution time of software, such as operating system and memory caches events. Taking into account these

X. Li et al. (Eds.): SEAL 2008, LNCS 5361, pp. 61–70, 2008.
© Springer-Verlag Berlin Heidelberg 2008

factors is difficult, and so compilers usually focus on optimising localised areas of code, rather than restructuring entire functions.

More sophisticated optimisations can be applied if we take into account the probability distribution of the *usage* of the software. For example, if a function takes an integer input and if we know that this input will usually be positive, this information could be exploited by optimising the software for positive input values.

In this paper we present a novel framework based on evolutionary optimisation techniques for optimising software. Given the code of a function as input to the framework, the optimisations are performed at the program level and consider the probability distribution of inputs to the program. To our best knowledge, we do not know of any other system that is able to automatically perform such optimisations.

Our approach uses Multi-Objective Optimisation (MOO) and Genetic Programming (GP) [1]. In order to preserve semantic integrity whilst improving efficiency, we apply two sets of test cases. The first is co-evolved with the program population [2] to test the semantics of the programs. The second is drawn from a distribution modelling expected input, and is used to assess the non-functional properties of the code. The original function is used as an oracle to obtain the expected results of these test cases.

Evolving correct software from scratch is a difficult task [2], so we exploit the code of the input function by seeding the first generation of GP. The first generation will not be a random sample of the search space as is usually standard in GP applications, but it will contain genetic material taken from the original input function. Note that this approach is similar to our previous work on *Automatic Bug Fixing* [3], in which all the individuals of the first generation were equal to the original incorrect software, and the goal is to evolve a bugfree version. A similar approach has also been previously taken in attempting to reduce the size of existing software [4].

We present a preliminary implementation of the novel framework, and we validate it on a case study. We then apply systematic experimentation to determine the most important factors contributing to the success of the framework. Although our prototype is still in an early stage of development, this paper gives the important contribution of presenting a general method to automatically optimise code using evolutionary techniques. We are also able to provide some guidance to other practitioners in applying such an approach, based on our analysis of empirical results.

The paper is organised as follows. Section 2 describes in detail all the components of the novel framework, whereas Section 3 presents our case study. Section 4 describes our results and Section 5 suggets further work.

2 Evolutionary Framework

An overview of our framework is given in Figure 1. The framework takes as input the code of a function or program, along with an expected input distribution, and then it applies GP to optimise one or more non-functional criteria. Note that in our experimentation, we chose to parameterise the use of MOO and Co-evolution in order to assess their impact on the ability of the framework to optimise non-functional properties of the software. The main differences from previous GP work are how the first generation is seeded, how the training set is used and generated, the particular use of