Neurocomputing 123 (2014) 381–389

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

Multi-objective adaptive evolutionary strategy for tuning compilations

Antonio Martínez-Álvarez [a,*], Jorge Calvo-Zaragoza [a], Sergio Cuenca-Asensi [a], Andrés Ortiz [b], Antonio Jimeno-Morenilla [a]

[a] Computer Technology Department, University of Alicante, Carretera San Vicente del Raspeig s/n, 03690 Alicante, Spain
[b] Communications Engineering Department, University of Málaga, Málaga, Spain

ARTICLE INFO

Article history:
Received 26 March 2013
Received in revised form
3 July 2013
Accepted 7 July 2013
Communicated by A. Prieto
Available online 19 August 2013

Keywords:
Tuning compilations
Evolutionary search
Genetic algorithm
Adaptive strategy
Multi-objective optimization
NSGA-II

ABSTRACT

Tuning compilations is the process of adjusting the values of a compiler options to improve some features of the final application. In this paper, a strategy based on the use of a genetic algorithm and a multi-objective scheme is proposed to deal with this task. Unlike previous works, we try to take advantage of the knowledge of this domain to provide a problem-specific genetic operation that improves both the speed of convergence and the quality of the results. The evaluation of the strategy is carried out by means of a case of study aimed to improve the performance of the well-known web server Apache. Experimental results show that a 7.5% of overall improvement can be achieved. Furthermore, the adaptive approach has shown an ability to markedly speed-up the convergence of the original strategy.

1. Introduction

One of the most important and fundamental issues of any branch of engineering is optimization. For example, the optimization of the resources and the costs in the manufacture of a product, the optimization of the development time of a solution or the optimization of the most important features of a product. Industry solutions that succeed are those that appear before, solve the problem and also minimize the related costs.

When trying to optimize the performance of a software application, many involved elements and conditions must be taken into consideration. In this context, compilers suppose a decisive factor in the optimization. Compilers currently offer a large number of optimization options. However, this capability is never fully exploited as it involves a comprehensive understanding of the underlying computer architecture, the target application and the operation of the compiler. The selection of the most convenient compiler options to improve a specific target (e.g. execution time, code size, cache L2 misses, etc.) represents a very complex task due to several reasons: modern most-used compilers such as GCC [1], Clang [2] or ICC [3] provide a large option set that can change the features of the application, some option combinations can change the normal execution (e.g. code vectorization could relax the accuracy of floating point operations) and dependencies amongst options cannot be known in advance as they also depend on the target application. Thereby obtaining the best combination by brute-force is unfeasible in terms of computational cost.

Aforementioned reasons lead us to propose a strategy that speeds up the exploration of compilation space and produces good solutions in an affordable time. Several authors have proposed different approaches to do this search. Pinkers et al. [4] considered the compiler as a black box in which nothing about the inner workings is known. They selected only a subset of compiler options using orthogonal vectors. ACOVEA [5] used a genetic algorithm to find the best options for speed-up programs compiled using GCC. Bashkansky and Yaari [6] proposed a framework (ESTO) to obtain suboptimal compilations using a genetic algorithm.

All previous proposals try to optimize a single criterion as a result of the compilation (usually the execution time) and ignore other features that may be equally important in relation to performance. The performance improvement of an application can be characterized by various technical criteria and design constraints that must be satisfied simultaneously and optimized as far as possible. Occasionally, these criteria may conflict and result in a mutual worsening (e.g. performance and power consumption in embedded systems).

In this paper, a genetic algorithm to explore the solution space is also proposed and, in addition, we include two significant contributions.

First, our approach uses a Multi-Objective Optimization (MOO) strategy to avoid this problem. These strategies consist in searching a set of optimal solutions for a set of criteria. Although the

* Corresponding author. Tel.: +34 965903400x3043; fax: +34 965909643.
E-mail addresses: amartinez@dtic.ua.es (A. Martínez-Álvarez), jcalvo@dtic.ua.es (J. Calvo-Zaragoza), sergio@dtic.ua.es (S. Cuenca-Asensi), aortiz@ic.uma.es (A. Ortiz), jimeno@dtic.ua.es (A. Jimeno-Morenilla).

0925-2312/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.neucom.2013.07.036

combination of MOO schemes with genetic algorithms has been widely used before [7–9], the proposed application for tuning compilations is innovative.

Second, knowledge of the problem domain allows us to improve the execution of the genetic algorithm by including custom genetic operators (also known as Problem-Specific Genetic Algorithms [10,11]). In the context of tuning compilations, there are two main aspects that we have taken into account to improve the strategy:

1. Compilations errors may occur due to incompatibilities between options. These compilations must be avoided in applications where this process is very expensive in terms of time. Some of these inconsistencies cannot be known in advance as they depend on the specific target application.
2. For each pair application/processor generally exists a set of options that can dramatically improve the performance. Learning this knowledge could be used to rapidly guide the genetic algorithm towards the most profitable solutions.

The inclusion of these features in the genetic algorithm improves both the quality of the results and the speed of convergence.

1.1. Background

Compilation process is strongly related to the target architecture. The available compilation options can alter the functioning of the system, as well as the interaction with the operating system, e.g. increasing or decreasing the number of context switches and cache misses, phenomena that can directly affect performance.

Performance of current microprocessors, with their complex pipelines and integrated data and instruction cache levels, is highly dependent on the compiler and its ability to structure the code for optimal performance. Obtaining the optimal program structure and scheduling is a complex process which is specific to each architecture, leading to large differences in performance depending on the employed optimization techniques. This task is carried out to the extreme in VLIW (Very Long Instruction Word) and EPIC (Explicitly Parallel Instruction Computing) architectures such as *Itanium* or *Itanium 2*. These microprocessors delegate the instruction scheduling to the compiler in order to reduce the complexity and free up space. In these cases, the compiler must statically determine the structure and exploit the parallel architecture to optimize the performance [12].

Besides the above stated, it should be noted that optimization options from modern compilers do not work in an atomic way, i.e. its influence varies depending on other modifiers. Because of this fact, the task of adjusting the compilation to take the maximum advantage of the system, has an enormous complexity, even with a full knowledge of the process. Although some compilers include predefined optimization levels (e.g. `-O0`…`-O3` option group for GCC, Clang or ICC), in most applications they are far from optimal.

On the other hand, the use of some optimization techniques can produce adverse effects. For example, function in-lining can make a program run faster by avoiding the time cost of routine calls. However, in-lining overuse can result in a very large program which directly affect the instruction cache misses and therefore the final performance [13]. Due to these circumstances we propose a strategy which combines a multi-objective optimization scheme for optimizing conflicting goals, with a genetic algorithm for speeding up the exploration of the search space.

In the next section the proposed strategy for tuning compilations is described. In Section 3, the specific-problem genetic operators are presented and discussed. Section 4 presents a case of study to evaluate the strategy whose results are drawn in Section 5. Last section concludes this work and discusses directions for future research.

2. Multi-objective optimization strategy for tuning compilations

The problem of getting the best option selection in the compilation process has been presented. Considering the number of possibilities to be taken into account, the problem cannot be solved by exploring the entire solution space. Therefore, the proposed strategy uses a genetic algorithm as search engine. Genetic Algorithms (GA) are a stochastic search technique inspired by the theory of evolution and belongs to the group of techniques known as Evolutionary Algorithms (EAs). These techniques are based on imitating evolutionary processes as natural selection, crossover and mutation.

A GA operates on a set of individuals and each one represents a possible solution to the problem. Each individual is encoded by its chromosome, comprising a number of genes which represent parts of the solution. These individuals are initialized randomly and better solutions are obtained through crossover and mutation operators. Subsequently, individuals are evaluated and selected so that only those who codify the best solutions can survive. At the end of the process, a set of solutions can be extracted from the surviving population.

For the problem presented here, the best solutions are those that most optimize the target application. Optimization during compilation consists in setting the compiler options to improve certain features of the program without altering the results, i.e., maintaining the correctness. Typically, the most common goal is to reduce both execution time and code size. However, many of the compiler options reduce the execution time by increasing the size of code, and vice versa. Therefore, finding the best trade-off is far from being trivial. Moreover, these are not the only conflicting criteria that can be taken into account, especially if the scope of the problem is not for general purpose machines. For example, in embedded systems there are other factors like power consumption, security and fault tolerance. In these situations, when assessing the quality of a solution is unclear, several approaches can be adopted. In this work, a multi-objective optimization scheme has been implemented.

2.1. Genetic algorithm

In our approach (Fig. 1), the chromosome of each individual (G) represents a possible compilation. As can be noted, every gene (G_i) can only take two possible values (0 or 1). Moreover, we consider two different types of optimization options. The first one is defined by a compiler flag such as `-finline-functions-called-once` which is encoded using a single gene that can be enabled, taking the value 1, or disabled, taking the value 0. The second type is defined by options taking values within a set. This is the case of the GCC generic optimization level `-OX` where $X \in \{0, 1, 2, 3, s\}$. This information is encoded using N genes, where N is the number of possible values (5 in the given example). Only the gene representing the chosen value from set will take the value 1, while the rest remain 0.

To provide flexibility, our system is not restricted to any given compiler. Thus, taking a selected compiler (GCC in our case study) the set of compiler options and its possible values have to be provided to the system by means of a configuration file. The population is initialized randomly, but specification of initial individuals is allowed to take advantage of prior knowledge.

The process consists in five stages. In the initialization step, a random new population is created by specifying the compiler