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*

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, an 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% overall improvement can be achieved. Furthermore, the adaptive approach has shown an ability to markedly speed-up the convergence of the original strategy.

Keywords: Tuning Compilations, Evolutionary Search, Genetic Algorithm, Adaptive Strategy, Multi-Objective Optimization, NSGA-II

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

*Corresponding author

Email addresses: amartinez@dtic.ua.es (Antonio Martínez-Álvarez), jcalvo@dtic.ua.es (Jorge Calvo-Zaragoza), sergio@dtic.ua.es (Sergio Cuenca-Asensi), aortiz@ic.uma.es (Andrés Ortiz), jimeno@dtic.ua.es (Antonio Jimeno-Morenilla)

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 can not 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 an 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. Guy Bashkansky and Yaakov 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.

Firstly, 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 combination of MOO schemes with genetic algorithms has been widely used before [7, 8, 9], the proposed application for tuning compilations is innovative.

Secondly, knowledge of the problem domain allow 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 can not 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 can not 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.

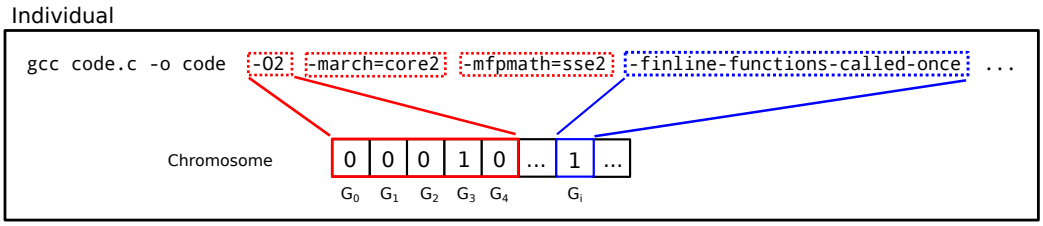

Figure 1: Example of an individual codification for GCC compiler

The process consists in five stages. In the initialization step, a random new population is created by specifying the compiler options. In the selection stage, the size of the population is controlled by eliminating additional individuals. New individuals are created through crossover operator. This operator takes two individuals and creates a new one by choosing uniformly genes from them. At the mutation stage, three fourths of the population are mutated. This value was established after showing the best trade-off between keeping the elite and covering the solution space in a preliminary experimental phase. For each gene, this process decides whether it changes or not the value using a low probability. Finally, in the evaluation stage each individual compiles the target application and evaluates the specified set of criteria. This process leaves a single fitness value for each criterion. At the end of this stage, the population is sorted using the MOO algorithm as it will be explained in the following section.

These last four steps are repeated until the algorithm reaches a stopping condition (e.g. number of iterations). Finally, the output of the algorithm consists of all surviving individuals.

2.2. Multi-Objective Optimization Scheme

In recent years various methods have been proposed to solve MOO problems. These methods can be classified into two groups: the former are relatively simple algorithms based on the Pareto front (NSGA [14], NPGA [15], VEGA [16] and MOGA [17]), which have fallen into disuse, and a second group of elitist algorithms that emphasize computational efficiency SPEA [18], SPEA2 [19], NSGA II [20], MOGLS [21], PESA [22], PESA II [23]). Currently, NSGA II (Non-dominated Sorting Genetic Algorithm II) is recognized as one of the most successful algorithms.

The NSGA II algorithm consists in classifying the individual in non-dominated Pareto fronts. Individuals classified in the first front (F_1) are

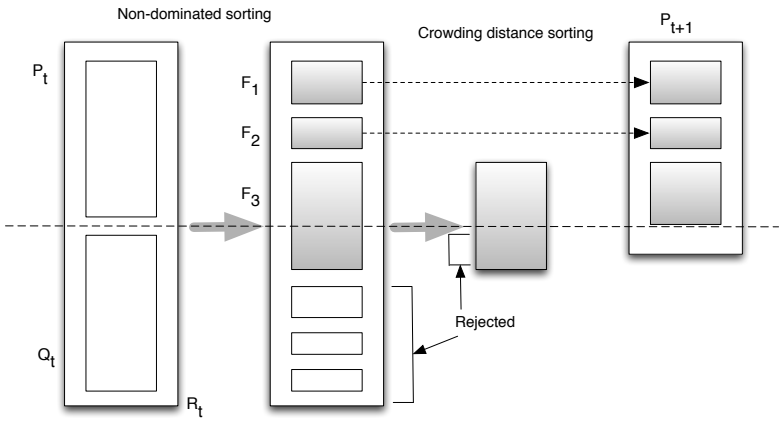

Figure 2: Population selection using NSGA II

those that are non-dominated. Individuals that belong to the second front are those that are non-dominated in absence of the first front, and so on. It is said that one solution X_i dominates other X_j if the first one is better or equal than the second in every single objective and, at least, strictly better in one of them.

When the fronts are built, individuals have to be sorted again in order to maintain a good spread of solutions. NSGA II uses the crowding distance to estimate the diversity value of a solution regarding the whole population. This way, individuals are evaluated based on its diversity within the dimension of each objective. The final crowding distance value depends on the sum of values corresponding to each objective.

After the execution of the NSGA II algorithm, the population is first sorted by non-dominated ranks and then, by the crowding distance (Fig. 2).

3. Adaptive Genetic Operation for Tuning Compilations

In this section, the usual running of the genetic algorithm is modified to be adapted to the specific domain it is dealing with by means of proposing new genetic operators. As it was mentioned in the Introduction, there are two aspects of the tuning compilations process that could be improved: (1) detection of the incompatibilities amongst compiler options can reduce the search space and therefore increase the efficiency of the strategy, (2) the inference of the most beneficial options can speed up the convergence of the genetic populations as it guide the algorithm to the best solutions.

3.1. Incompatible Genes Detection

Although some incompatibilities between the compilation options are reported by the compiler, the majority of them can not be *a priori* known because their dependence on the target application itself. Therefore, it is necessary for the algorithm to detect them while running. Since most compilations are successful, a large number of compatibilities can be verified quickly. For this work, the assumption that incompatibilities occur only between pair of options is adopted, once the individual ones has been ruled out previously by the user.

Two approaches have been mainly considered for the design of this module: a) comprehensive strategy: it only tags two genes as incompatible when there is no doubt about it, b) probabilistic strategy: it approaches the likelihood of being incompatible of every pair by using the successive results of previous compilations.

3.1.1. Comprehensive Strategy

This strategy consists in taking advantage of the many successful compilations to check the compatible pairs. Thus, when a compilation fails, the likelihood of knowing which pair of options has caused it is high. Let n be the number of options (i.e. genes), G the chromosome of an individual (G_i refers to the ith gene), and D a $n \times n$ matrix filled with three possible values ($Incompatible, Compatible, Unknown$), each one to indicate a concrete knowledge of every pair of genes. The initial value of every matrix element is $Unknown$. This first algorithm is shown below.

When a compilation is successful, $check(individual)$ returns $True$ and all enabled options are tagged as $Compatible$ amongst themselves. When the compilation is erroneous, it is checked if it can be known which pair (G_i, G_j) of genes is causing the incompatibility. This is done by checking if the rest of pairs are $Compatible$. The main disadvantage of this strategy is that it does not assure the convergence since it needs specific conditions to detect a incompatibility.

3.1.2. Probabilistic Strategy

The probabilistic strategy is based on using each compilation (individual) to approximate the solution in terms of probability. P_{ij} represents the probability that the ith gene is incompatible with the jth gene. For this purpose, an $n \times n$ matrix is filled with these probabilities. By design, this matrix must be symmetric ($P_{ij} = P_{ji} \quad i, j \in \{1, \ldots, n\}$) at the end of the execution.

Algorithm 1 Comprehensive strategy

Require: $individual; D \in \mathbb{R}^{n \times n}$
  if $check(individual)$ **then**
    for all $G_i = 1 \in individual$ **do**
      for all $G_j = 1 \in individual$ **do**
        $D_{ij} \leftarrow Compatible$
      end for
    end for
  else
    for all $G_i = 1 \in individual$ **do**
      for all $G_j = 1 \in individual \wedge D_{ij} = Unknown$ **do**
        $inc \leftarrow TRUE$
        for all $G_k = 1 \in individual \wedge G_l = 1 \in individual \wedge k, l \neq i, j$ **do**
          if $D_{kl} \neq Compatible$ **then**
            $inc \leftarrow FALSE$
          end if
        end for
        if $inc = TRUE$ **then**
          $D_{ij} \leftarrow Incompatible$
        end if
      end for
    end for
  end if

Algorithm 2 Probabilistic strategy

Require: $individual; P \in \mathbb{R}^{n \times n}; U, \alpha \in \mathbb{R}$
  if $check(individual)$ **then**
    for all $G_i = 1 \in individual \wedge G_j = 1 \in individual$ **do**
      $P_{ij} \leftarrow 0$
    end for
  else
    $M \leftarrow \left(\frac{N - |G_i = 1|}{N-2}\right)^{\alpha}$
    for all $G_i = 1 \in individual$ **do**
      for all $j \neq i \wedge P_{ij} > 0$ **do**
        if $G_j = 1$ **then**
          $P_{ij} \leftarrow P_{ij} + (1 - P_{ij}) \cdot M$
        else
          $P_{ij} \leftarrow P_{ij} + (0 - P_{ij}) \cdot M$
        end if
      end for
    end for
  end if
  for all P_{ij} with $i \leq j$ **do**
    $P_{ij}, P_{ji} \leftarrow \frac{P_{ij} + P_{ji}}{2}$
    if $P_{ij} \geq U$ **then**
      $Incompatibles \leftarrow Incompatibles \cup (i, j)$
    end if
  end for
  return Incompatibles

The Algorithm 2 shows the following strategy. If the compilation successes, $check(individual)$ returns $True$, and every probability between each pair of enabled options is set to 0. When the compilation fails, a factor M is calculated as a function depending on the number of enabled options. This factor M, which will be 1 when there are only two enabled options and 0 when every single option is enabled, is used to close to 1 the probabilities of each pair of enabled options and to 0 for each pair in which one is active and the other is not. To ensure the symmetry of matrix P, the values P_{ij} and P_{ji} are forced to be the average of them.

The advantage of this strategy is that it attempts to approximate the solution in each iteration. In contrast, it does not ensure that incompatibilities are truly such. It leaves the responsibility in the choice of the parameters α and U.

As none of the above strategies provides a solution that meets our needs, an hybrid approach has been adopted: the main algorithm will assume both strategies simultaneously, leaving the final decision on the comprehensive one. Meanwhile, the probabilistic strategy will be used to propose genes that show evidence of being incompatible. This information will be used in an additional operator of the genetic algorithm that will inject these individuals in the current population to verify if they are really incompatible. The detection of incompatibilities must be executed after the evaluation stage, when it is known whether a particular individual has produced a successful compilation or not.

As a usual process of acquiring knowledge, the algorithm must ensure that inconsistent individuals are avoided. For this reason, before the evaluation stage, individuals are checked to know if someone is going to produce a compilation error. If it is found any individual that has two options that have been declared incompatible with each other, the individual will be tagged as wrong without performing the compilation process. This decision has been taken to produce a minimal effect on the conventional operation of the genetic algorithm.

3.2. Option Influence Study

To provide to the original strategy an ability of learning, a new algorithm to infer the influence of each gene in the individual quality is proposed. The design of such algorithm must consider the following constraints:

1. Each option affects differently and isolated from the rest.

2. Disabled options do not affect.
3. The influence can be positive or negative.
4. Each option affects differently to each criterion.

The inference of this influence must be commensurate to the value of fitness obtained by different individuals during the algorithm. It exists two ways of quantifying the goodness of a given individual fitness value: (1) the user defines the expected value or the range in which it is expected to be found, (2) a range and a mean is assessed by the successive obtained values. The former approach has the problem of quantifying the range because if the estimation is pessimistic all individuals would seem good enough. An excessive reduced range would produce the opposite effect. Additionally, the user would need to have a prior knowledge that is barely possible. The latter option provides a way to alter the learning consistently with the obtained values, without requiring prior information. Thus the second option was chosen.

The proposed algorithm consists of two phases: accumulation and normalization. In the accumulation phase, illustrated in Algorithm 3, the strategy stores a value for each option accordingly to the fitness value and its difference from the mean. This value is divided by the maximum possible difference. Subsequently, this calculation is accumulated for each enabled option.

Algorithm 3 Influence study: accumulation

Require: $individual; \overrightarrow{influence} \in \mathbb{R}^n; fitness, mean, best, worst \in \mathbb{R}$
  $M \leftarrow (fitness - mean)/\max\{best - mean, mean - worst\}$
  for all $G_i = 1 \in individual$ **do**
    $influence_i = influence_i + M$
  end for
  return $\overrightarrow{influence}$

On the other hand, the normalization phase occurs after each iteration and it is designed to obtain a value for each option. This value can be seen as the conditional probability $P(G_i|I_o)$, i.e. the probability that a given gene (G_i) is activated assuming the optimum individual (I_o). Thus, a gene with a probability of 0.5 would be a valueless gene, a higher value would indicate that this gene has a positive influence, and a smaller value a negative influence.

With the vector obtained in the accumulation phase the influence vector $P(G|I_o)$ is calculated (Algorithm 4). Each option gets its probability accordingly to the value in the accumulation vector. The operation will be normalized so that the value remains in the interval $[0, 1]$.

Algorithm 4 Influence study: normalization

Require: $individual; \overrightarrow{influence}, \overrightarrow{P} \in \mathbb{R}^n$
  $best \leftarrow \max_{i=0...n-1}\{influence_i\}$
  $worst \leftarrow \min_{i=0...n-1}\{influence_i\}$
  for all $G_i \in individual$ **do**
    if $influence_i > 0$ **then**
      $P_i \leftarrow 0.5 - 0.5 \cdot (influence_i/best)$
    else
      $P_i \leftarrow 0.5 + 0.5 \cdot (influence_i/worst)$
    end if
  end for
  return $\overrightarrow{P}$

Calculation of the influence must be carried out after the evaluation stage, when the values of fitness are obtained. Then the influence vector is used to guide the mutation process to generate individuals whose genes are those with the most positive influence. This is carried out by means of a control function f_c taking values in the interval $[0, 1]$, to avoid the effect of local minimum convergence during the first iterations. The weight of the vector increases with the number of iterations as the estimation become more reliable. The mutation process is set as shown in the Algorithm 5, where P_{mut} is the original mutation probability, $rand()$ is a pseudo-random number in the interval $[0, 1]$, P_i is the obtained influence for the gene ith and $f_c(x)$ the control function, whose value is based on the current iteration x.

Finally, to consider the multi-objective optimization, the approach is to manage as many influence vectors as defined fitness functions, and combining these vectors before the mutation by means of the average operator.

4. Case of Study: Optimization of Apache Web Server

The exponential demands for high performance web servers is a trend which is gaining importance in the world of information and business-oriented services [24]. In this section, the proposed strategy is applied in an effort

Algorithm 5 Custom mutation operator

Require: $individual; P_{mut}, P_i, x \in \mathbb{R}$
  for all $G_i \in individual$ **do**
    $P \leftarrow (1 - f_c(x)) \cdot P_{mut} + f_c(x) \cdot P_i$
    if $G_i = 0 \wedge rand() < P$ **then**
      mutate(G_i)
    else if $G_i = 1 \wedge rand() > P$ **then**
      mutate(G_i)
    end if
  end for

to improve the performance of the Apache web server running on a Linux machine. As valuable objectives involving performance we have selected the following 4 metrics of interest: number of operating system context switches, second level (L2) cache misses, web server throughput and mean time per request.

Modern web servers devote most of the execution time to the network operating system stack. Therefore, the network stack has a great influence on the global performance. In these applications, the number of cache misses is usually high due to the fact that the Linux TCP/IP stack does not fit in usual sized L2 caches. Furthermore, if pages with random data (which can avoid the microprocessor predictors) are used, L2 cache misses can increase as well. As DRAM latency is usually over a hundred clock cycles, these cache misses greatly reduce the throughput of the processor. Therefore, all the mentioned criteria must be taken into account to measure the overall performance.

The number of operating system context switches was calculated using Linux `/proc` filesystem utilities, the number of second level (L2) cache misses was calculated using *OProfile* [25] and finally, the web server throughput and mean time per request were calculated using *Apache Benchmark* (*AB*). *AB* allows to perform load tests with different configurations and it provides a range of useful performance indicators, such as the number of requests processed per second or the number of failed requests. OProfile is a fine-grained code profiling tool which consists of a Linux kernel driver, a daemon and many reporting tools. By means of sampling, it collects data from the hardware performance counter registers within defined time intervals. Unlike other tools, OProfile can profile the whole system without the need to modify

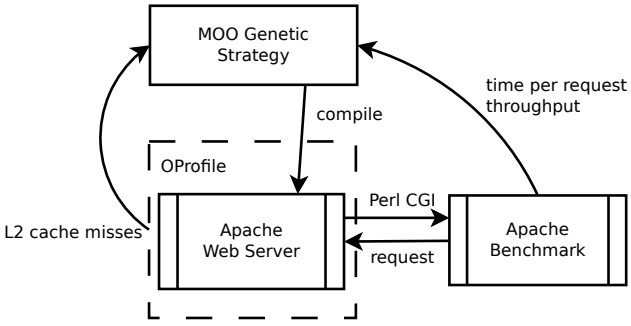

Figure 3: Scenario of the Apache case of study

the target application. Finally, the context switches were obtained from Linux */proc/stat* reports. Figure 3 represents the scenario of the case of study.

To take into account the worst-case scenario, avoiding the CPU predictors is mandatory [26]. Thus, the requested page was established as a Perl CGI script that generates random real numbers printed on a single line. This test also allows us to have more precise measurements since it minimizes I/O usage.

This case of study consist in two parts. Firstly, a set of the better solutions, based of aforementioned metrics, are obtained from the execution of the original genetic algorithm. These results allow us to evaluate the main objective of our strategy: the overall improvement of the target application. Secondly, the execution is repeated with the custom genetic algorithm operation developed in Section 3 to show the speed-up that it can provide.

The experimental setup is composed of Intel Core 2 Duo 2.3GHz machines with 4 GiB of RAM and a Realtek RTL8169 based gigabit network interface controller (NIC), running Debian Linux with the 3.2.0-1-amd64 kernel and GCC 4.6.2. The NIC maximum transmitted unit (MTU) was configured to the maximum allowed value of 7000 to optimize the bandwidth. Apache version 2.2.21 was used in our tests. Tests have been performed with page lengths of 2 KiB.

4.1. Strategy Evaluation

The overall optimization process over 145 generations with 34 individuals per each took about 6 days. The results are expressed with sets of points facing each pair of criteria. The GCC default optimization levels (`-O1, -O2, -O3`) are included in this set in order to see the achieved improvement. As a

result of the optimization process, we obtain the Pareto front which summarizes the solutions found by the multiobjective optimization process. Thus, figures 4,5,6,7,8, and 9 represent the calculated solutions to any pair of the 4 given objectives (Number of L2 global misses, Number of context switches, Throughput and Mean time per request). The Pareto front is showed in the given figures and represent the best solutions (individuals) in terms of the trade-off of any given objective. The individuals belonging to the Pareto front are identified by means of blue dots.

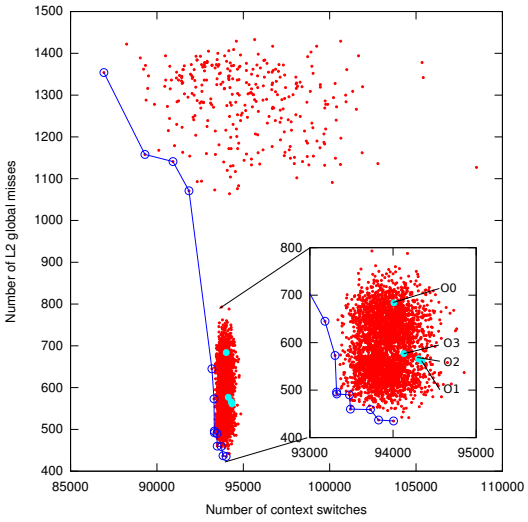

Figure 4: L2 cache misses against context switches

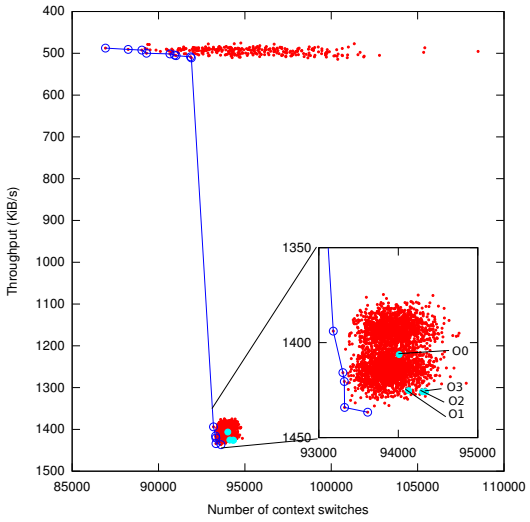

Figure 5: Throughput against context switches

There are two different groups consistently throughout all results. Figure 4 shows a point cloud representing the set of solutions obtained throughout the execution based on L2 cache misses and the number of context switches. The solutions belonging to the Pareto front range from individuals with a low number of context switches and a high amount of L2 cache misses to more balanced individuals. The latter shows a significant reduction of cache misses compared to the predefined optimization levels.

Figure 5 shows the solutions in terms of throughput (in KiB/s) and context switches. Similarly to the previous graph (Figure 4) there are two different groups, more so in this case since the groups are more compact. Figure 6 represents the time per request and the number of context switches. The behavior is similar in both cases and express the inverse proportionality between those two criteria. There are differences however, because of the time

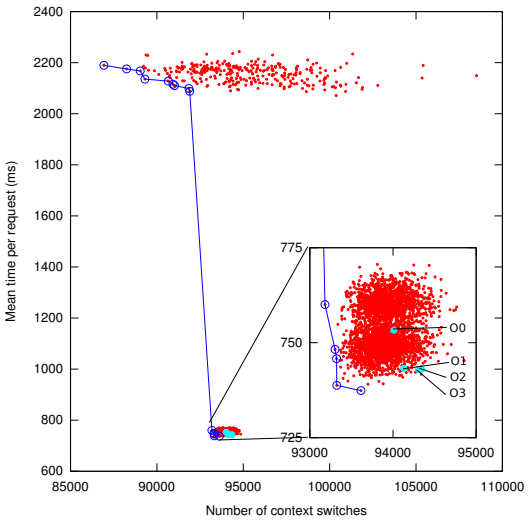

Figure 6: Time per request (mean) against context switches

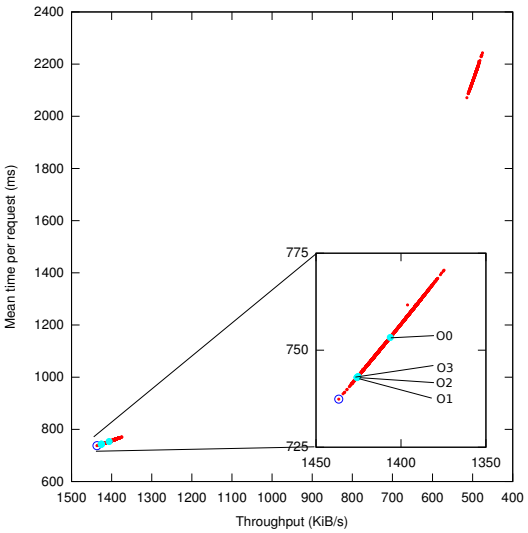

Figure 7: Time per request (mean) against throughput

per request is a mean value and the relationship is not necessarily linear (Figure 7).

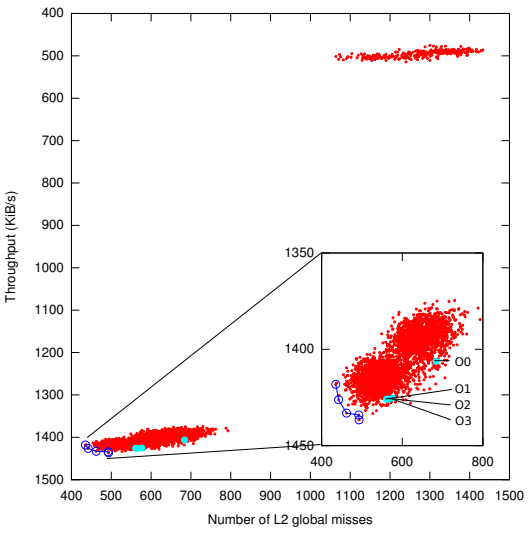

Figure 8: Throughput against L2 cache misses

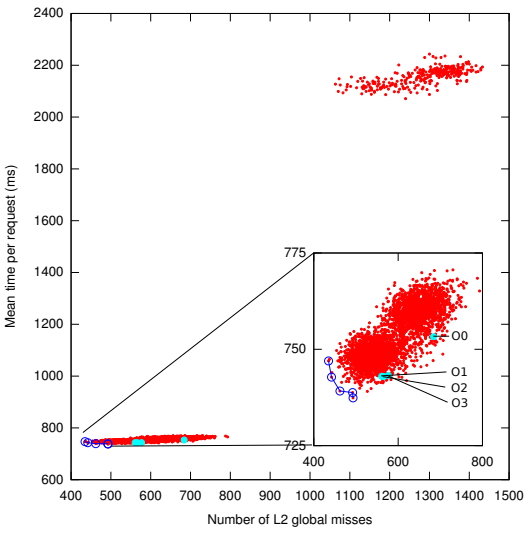

Figure 9: Time per request (mean) against L2 cache misses

On Fig. 8 the *x-axis* represents the number of L2 cache misses while the

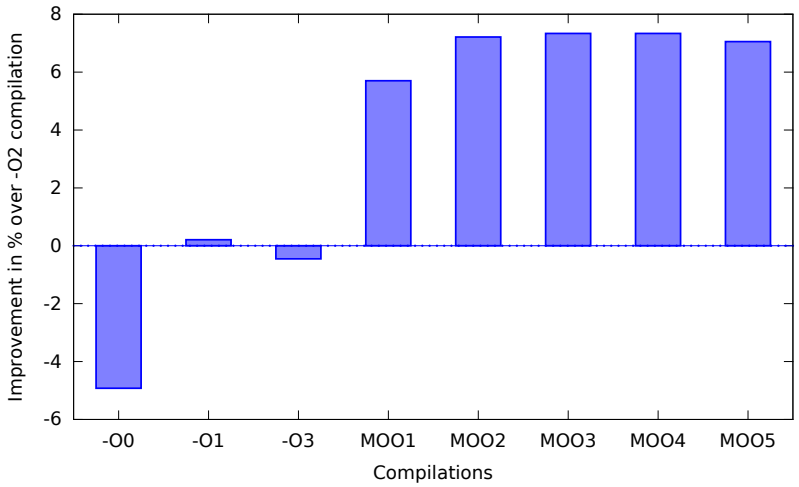

Figure 10: Joint improvement with respect to the GCC `-O2` optimization

y-axis represents the throughput. A relation can be seen between these two criteria: individuals with the highest number of cache misses get a much lower throughput. The predefined optimization levels, nevertheless, produce a good throughput despite having a worse result in cache misses. The solutions belonging to the Pareto front have a marginally higher throughput and are significantly better in terms of cache misses. The situation is similar when comparing the mean time per request and the L2 cache misses (Fig. 9).

Since our goal is to find compilations which most improve all criteria at the same time, Fig. 10 shows the mean over all criteria for the best five individuals. This average can be used as a measure of the joint improvement. `-O2` compilation has been established as the baseline as it is activated by default in Apache compilation. As it can be seen, `-O0`, `-O1` and `-O3` points are dominated by the best solutions reached by the MOO strategy. Furthermore, `MOO2`, `MOO3` and `MOO4` individuals offer an overall improvement higher than 7% (7.5% in the case of `MOO3`).

4.2. Custom Operators Evaluation

To manage consistent values in the comparison, the same experiment is repeated after adding to the strategy the developed improvements. The necessary parameters have been established as follows: $\alpha = 3$, $U = 0.9$ (Algorithm 2), $P_{mut} = 0.05$ and $f_c(x) = \frac{x}{200}$ (Algorithm 5). To handle

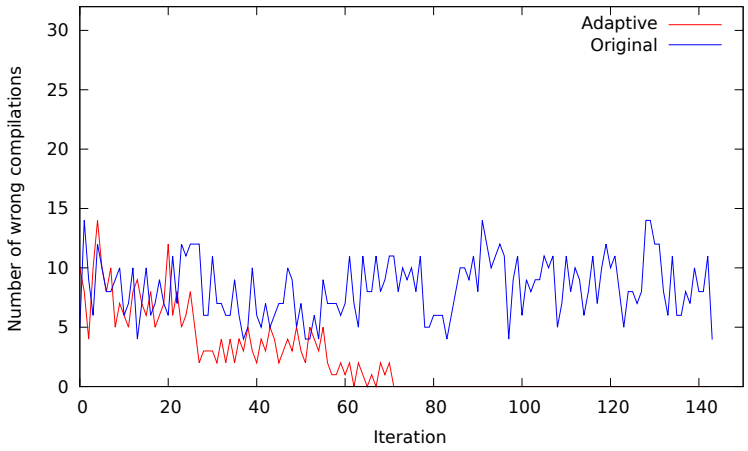

Figure 11: Number of wrong compilations for each iteration

multi-objective optimization, the final vector of influence is calculated as the average of the vectors of influence of each single objective.

The test has been extended for the same time as the original, performing this time a total of 200 iterations (38% higher). During the test 4 incompatibilities were detected (eg. GCC options `-fsched-stalled-insns-dep` and `-fcprop-registers` are revealed as incompatible), in iterations 27 (two of them), 55 and 70. Figure 11 shows the number of wrong compilations produced in each iteration. Those produced by the original strategy are shown in blue and those produced by the improved strategy in red. tt

While in the original test an 27% of the individuals had incompatibilities, with the custom operator this percentage has dropped to 12%. It should be noted that all wrong compilations have been eventually avoided. Quantitatively, this difference allowed the strategy to run 55 iterations more in a similar time period. Thus the new strategy has been much more effective with its execution time.

On the other hand, to measure the goodness of the influence inference, Figure 12 shows the *fitness* mean of the population in each iteration on the criterion of context switches. The population value of the original strategy is marked in blue and the population of the improved one in red. It can be seen that the proposed operators produce a significant acceleration of convergence. The algorithm achieves the best value obtained during the original test around the 100th iteration. Same meaning does Figure 13 on

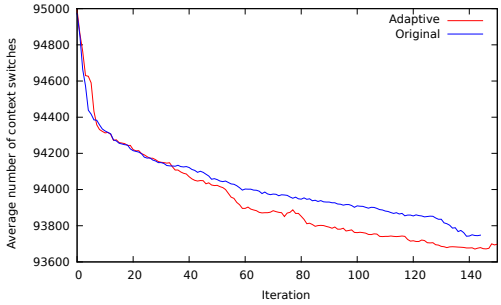

Figure 12: Evolution of context switches

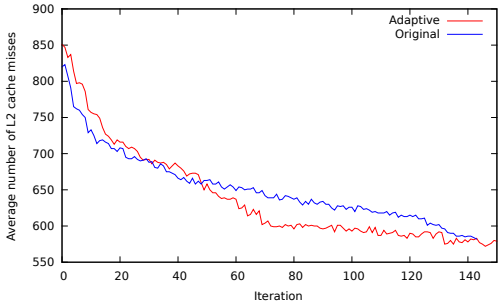

Figure 13: Evolution of number of L2 cache misses

L2 cache misses. In this case the convergence expands to the 120th iteration although it is clearly seen how the values are improving along the execution.

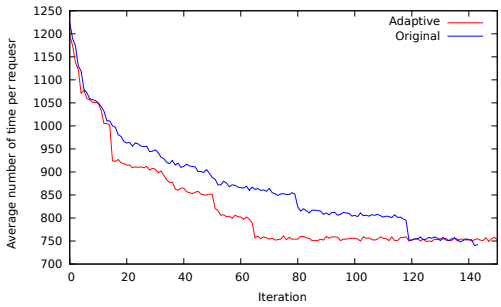

Figure 14: Evolution of time per request

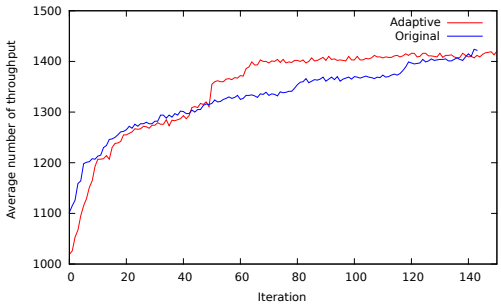

Figure 15: Evolution of throughput

On Fig. 14 is drawn the mean population value in each iteration on the time per request criterion. This time, the acceleration of convergence is much more visible and the new strategy achieves the best value in the halfway iteration. The same phenomenon on transfer rate can be observed on Fig. 15, where the improved strategy converges at iteration 75. The analysis of these graphs yields the conclusion that promotion of the best compiler options produces a visible acceleration in the convergence of the genetic algorithm, especially for both time per request and throughput criteria.

5. Conclusions

This paper presents an optimization strategy that guides the search of the best combinations of compiler options, to maximize the performance of an

application of interest. Unlike previous works, our strategy allows taking into account several metrics equally important for improving the overall performance. Results show that the combination of a GA with a MOO algorithm, as NSGAII, performs well dealing with multiple criteria satisfaction.

Experimentation proves that in the case of tuning compilations it is possible to take advantage of the prior knowledge about the usual running of the compilers.

This knowledge leads us to modify the conventional operation of the GA to make a better use of its searching. Specifically, this paper has addressed the problem of detecting incompatibilities between compiler options (especially those not reported by the compilers). The study of the influence of each option in the performance of the application guides the system towards individuals with the most favorable options. A hybrid strategy is proposed to deal with the incompatibility gene problem. It detects inconsistencies in which there is no room for doubt and empirically tests those in which there are some evidences. Regarding the study of the influence, it has been proposed a two-stage algorithm which collects information from the successive obtained values and it then gives a probability value for each option considered.

In our case study, which is focused on optimizing the operation of the well-known Apache web server, we found a limited room for improvements with respect to the default compilation in each criterion separately. Nevertheless, our strategy has demonstrated an ability to find an overall improvement up to 7.5% for 2KiB web pages. The inclusion of custom operators in the GA has shown a significant reduction of the space of solutions and a marked acceleration of the speed of convergence.

This paper opens several lines of future work related to the study of the influence of compilers options in the improvement of different application domains.

References

[1] GCC, GNU Compiler Collection, `http://gcc.gnu.org` [Last accessed: 06/25/2013], 1987.

[2] CLang, A C language family frontend for LLVM, `http://clang.llvm.org/` [Last accessed: 06/18/2013], 2005.

[3] ICC, Intel C++ Compiler, `http://software.intel.com/` [Last accessed: 06/18/2013], 2002.

[4] R. Pinkers, P. Knijnenburg, M. Haneda, H. Wijshoff, Statistical selection of compiler options, in: Proceedings of The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, IEEE Computer Society Washington, Washington, DC, USA, 2004, pp. 494–501.

[5] Scott Robert Ladd, Project ACOVEA, 2007. `http://freecode.com/projects/acovea` [Last accessed: 06/18/2013].

[6] G. Bashkansky, Y. Yaari, Black box approach for selecting optimization options using budget-limited genetic algorithms, in: Proceedings of SMART Workshop 2007 (HiPEAC). European Network of Excellence on High Performance and Embedded Architecture and Compilation, Ghent, Belgium, pp. 1–16.

[7] C. M. Fonseca, P. J. Fleming, Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization, in: Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo, CA, 1993, pp. 416–423.

[8] P. Poulos, G. Rigatos, S. Tzafestas, A. Koukos, A pareto-optimal genetic algorithm for warehouse multi-objective optimization, Engineering Applications of Artificial Intelligence 14 (2001) 737–749.

[9] R. Quiza Sardiñas, M. Rivas Santana, E. Alfonso Brindis, Genetic algorithm-based multi-objective optimization of cutting parameters in turning processes, Engineering Applications of Artificial Intelligence 19 (2006) 127–133.

[10] Y. Li, Y. Yang, M. Ma, R. Zhu, A problem-specific genetic algorithm for multiprocessor real-time task scheduling, in: Proceedings of the 3rd International Conference on Innovative Computing Information and Control, ICICIC '08., IEEE, Kaohsiung, Taiwan, 2008, pp. 186–186.

[11] E. Carrano, L. Soares, R. Takahashi, R. Saldanha, O. Neto, Electric distribution network multiobjective design using a problem-specific genetic algorithm, IEEE Transactions on Power Delivery 21 (2006) 995–1005.

[12] C. Dulong, R. Krishnaiyer, D. Kulkarni, An overview of the intel IA-64 compiler, http://noggin.intel.com/system/files/private/an-overview-of-the-intel-ia-64-compiler.pdf [Last accessed: 06/18/2013], 1999.

[13] F. P. Miller, A. F. Vandome, J. McBrewster, Inline Expansion: Compiler Optimization, Called Party, Branch (computer science), Algorithmic Efficiency, Return Statement, CPU Cache, Constant (programming), Compiler, Copy and Paste Programming., Alpha Press, 2010.

[14] N. Srinivas, K. Deb, Multiobjective optimization using nondominated sorting in genetic algorithms, Evolutionary Computation 2 (1994) 221–248.

[15] J. Horn, J. Horn, N. Nafpliotis, N. Nafpliotis, D. E. Goldberg, D. E. Goldberg, A niched pareto genetic algorithm for multiobjective optimization, in: Proceedings of the 1st IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, IEEE, Orlando, Florida, 1994, pp. 82–87.

[16] J. D. Schaffer, Multiple objective optimization with vector evaluated genetic algorithms, in: Proceedings of the First International Conference on Genetic Algorithms and Their Applications, Carnegie-Mellon University, Pittsburgh, PA, 1985, pp. 93–100.

[17] C. M. Fonseca, P. J. Fleming, Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization, in: Proceedings of the 5th International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, pp. 416–423.

[18] E. Zitzler, L. Thiele, Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach, IEEE Transactions on Evolutionary Computation 3 (1999) 257–271.

[19] E. Zitzler, M. Laumanns, L. Thiele, SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization, in: K. C. Giannakoglou, D. T. Tsahalis, J. Périaux, K. D. Papailiou, T. Fogarty (Eds.), Proceedings of Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems, International Center for Numerical Methods in Engineering, Athens, Greece, 2001, pp. 95–100.

[20] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 6 (2002) 182–197.

[21] A. Jaszkiewicz, On the performance of multiple-objective genetic local search on the 0/1 knapsack problem - a comparative experiment, IEEE Transactions on Evolutionary Computation 6 (2002) 402–412.

[22] D. W. Corne, J. D. Knowles, M. J. Oates, The pareto envelope-based selection algorithm for multiobjective optimization, in: Proceedings of the VI Conference on Parallel Problem Solving from Nature, Springer, Paris, France, 2000, pp. 839–848.

[23] D. W. Corne, N. R. Jerram, J. D. Knowles, M. J. Oates, M. J, Pesa-II: Region-based selection in evolutionary multiobjective optimization, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2001, Morgan Kaufmann Publishers, Granada, Spain, 2001, pp. 283–290.

[24] S. Sharifian, S. A. Motamedi, M. K. Akbari, A predictive and probabilistic load-balancing algorithm for cluster-based web servers, Applied Soft Computing 11 (2011) 970–981.

[25] John Levon, Philippe Elie, OProfile – A System Profiler for Linux, `http://oprofile.sourceforge.net/` [Last accessed: 06/18/2013], 2002.

[26] Intel, Intel 64 and IA-32 Architectures Optimization Reference Manual, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2011.