A User-Friendly Methodology for Automatic Exploration of Compiler Options

Haiping Wu Long Chen Joseph Manzano Guang R. Gao

University of Delaware
Department of Electrical and Computer Engineering
Newark, Delaware 19716, U.S.A
{hwu, lochen, jmanzano, ggao}@capsl.udel.edu

Abstract

This paper introduces a practical methodology for automatic exploring compiler optimization options. The strategy behind this methodology is to provide an intelligent mechanism, in which the compiler will automatically identify optimized combination of compiler options for a given application. The decision made by the compiler is based on user requirements on what aspects of the generated code are the most critical for the compiled application - through a specialized user interface. A set of pre-built databases of candidate optimized combinations of compiler options will help the compiler to make the right decision. This methodology will dramatically help users to get rid of the burden of fully understanding the compiler's inner structure and organization to make the most profitable combination of options. All this work is taken by the compiler and the infrastructure proposed in this paper.

This paper presents this methodology and describes its principles and technical mechanisms of the components.

Keywords: Compiler options, Evaluation Platform, Performance, Power, Code size

1. Introduction

Since the invention of the compiler, developers usually define the compiler-user interface using a methodology of compiler-driven option selection. It is the users' responsibility to find a good combination of compiler options to benefit their applications.

Modern compilers often have myriad options to control various aspects and degrees of optimization. For example, there are 60+ optimization options in the GNU Compiler Collection (GCC) C compiler [9]. This translates to more than 2^{60} possible combinations for any particular application!

As a result, finding an optimized combination of compiler options to benefit a particular user application presents a significant challenge to the compiler's users. To fully understand even a small subset of useful options often requires an in-depth knowledge of compiler's inner structure and organization - that a majority of users do not posses.

This situation is exacerbated on embedded systems. Unlike the general scientific computation domain where performance is the primary demand and a single combination of compiler optimization options is usually sufficient, more aspects need to be considered for applications in the embedded domain. For example, faster execution time, less power consumption and smaller code size are three primary aspects that are frequently interlaced together for applications in the embedded domain.

It is obvious that the traditional methodology of compiler-driven option selection is a major obstacle for users to fully take advantage of the rich optimization features that modern compilers have. There is a strong driving force to move toward finding new methodologies that can fill the gap between the myriad optimization features provided by compilers and the subset of features adopted by users.

In this paper, we introduce a methodology of exploring compiler optimization options, called User-friendly Methodology for automatic Exploration of Compiler Options or *UMECO* for short.

The strategy behind the methodology is to ask users for advice on what aspects of the generated code are the most important and what metrics should be considered critical for a given application - through a specialized user interface. Then, the compiler, based on the user's advices, as well as a set of pre-built databases of candidate optimized combinations of compiler options, will automatically identify the most profitable or optimized[1] combination of compiler op-

1 Term *optimized* is used to indicate a relative improvement of execution time, lower power consumption or a reduction in code size.

tions for the applications.

UMECO is a research project that is being developed at the University of Delaware. We are now working on several well-known embedded and multicore architectures, such as Intel XScale, IBM Cyclops-64 and CELL.

Our main contributions in this paper are:

- The idea of an automatic exploratory compiler option methodology and framework;
- The mechanisms and considerations of configuring evaluation platforms
- A strategy to choose adequate benchmarks
- Design issues and challenges when integrating a new methodology / framework with an existing compiler framework

The remainder of this paper is organized as follows. In Section 2, we outline the infrastructure of *UMECO*. Section 3 deals with the issues of configuring evaluation platforms. In Section 4, we discuss the methods of narrowing down the choice space from the whole combinational space of compiler options. The strategy to choose benchmark packages for measurement and testing is presented in Section 5. In Section 6, we discuss the issues of integrating *UMECO* inside a compiler framework. We present related work in Section 7. Finally, some conclusions are given in Section 8.

2. Infrastructure of the *UMECO*

2.1. The Components of *UMECO*

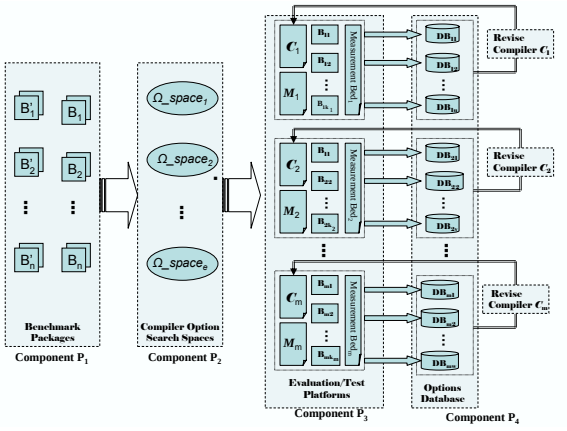

Figure 1. Components of *UMECO*

The *UMECO* consists of four components, as shown in Figure 1. Component P_1 consists of a group of benchmark packages. The packages on the right side (B_i, i =1,...,n) are used for measurement and the left side (B_i', i =1,...,n) are used for testing. Component P_2 consists of a set of narrowed space of combinations of compiler options. Each narrowed space(we name it as Ω*_space*) corresponds to a specific compiler and architecture pair. It is a trimmed down version of the combinatorial space[2]. Component P_3 is a group of evaluation platforms. Each evaluation platform consists of one architecture(hardware or simulator) and a compiler. Component P_4 consists of several set of databases which contains optimized combinations of compiler options. These databases will be built into the underlying compilers. A particular set of databases are created from a series of evaluations on a specific evaluation platform. The general methods of compiler implementation and other specific strategies, such as trade off between performance, power and code size, are also included in P_4.

2.2. The Working Principle of *UMECO*

The kernel component in *UMECO* is the evaluation platforms for measurement and testing. Each platform consists of a compiler, its corresponding toolchain, architecture that the compiler supports and a set of measurement tools.

A compiler-architecture pair determines the application domains that it supports and therefore determines what benchmark packages need to be chosen and used for measuring purposes. In other words, these chosen benchmark packages are used to create the databases, which will be used by the compiler to automatically identify optimized compiler options for the specific application domain. There is a set of standard benchmark suites for each application domain in *UMECO*. Measuring benchmark packages on a platform is a complex and time-consuming process. Depending on the number of elements in the Ω*_space*, a benchmark package may be measured several hundred times. Therefore, the selection of benchmark packages for measuring is another crucial consideration of *UMECO*. The benchmark packages for testing are used to verify the effects that *UMECO* has on the application. They come from real world applications.

Since the size of the combinatorial space is huge, a smart method must be used to narrow down this space to a smaller sub-space, or Ω*_space*. The measuring benchmark packages are ran using all of the combinations in this Ω*_space*. The combinations that produce an optimized results are chosen and stored into a database. For embedded systems, three factors (performance, power and code size) are considered as a whole; thus, a set of three databases are generated from

2 Represents the space of all possible combinations of compiler options

the measurement process. Each database corresponds to a factor specific measurement. For example, power measurement will generate a power specific database. Moreover, if the compiler supports several application domains, there is an individual set of databases for each application domain.

After the measurement phase finishes, the underlying compiler has to be revised to support the specific user advice options, by integrating the databases as one of its internal data structures.

After the compiler has been revised, the user can advice the compiler on what aspects mentioned above of the generated code should be given the top most importance, and also what domain this application belongs to. Then, the compiler automatically searches the optimized combination of compiler options from the built databases according to the user's advice.

3. Configuration of Evaluation Platforms

In this section, we present the concept of evaluation platforms formally, and describe the measurement methodologies that are crucial for the problem addressed in this paper.

3.1. Features of the Evaluation Platform

In *UMECO*, each evaluation platform includes a compiler and one architecture, which is supported by this compiler. For a specific architecture, there may be more than one compiler that could be used. In order to investigate the proposed problem, we have to construct the full mapping between compilers and the corresponding architectures, i.e, to enumerate all compiler-architecture pairs. Having all possible compiler-architecture pairs, the next step is to obtain the effects of different combinations of compiler options on each of those pairs, i.e. the execution time, power consumption, and code size. Moreover, after the compiler has been revised to provide the functionality of processing the user advice options, this functionality must be verified.

3.2. Measurement Techniques

In *UMECO*, measurement means the collection of information about performance, power and code size from a specific program on a real or simulation environment.

To collect such information, *UMECO* uses several methods. Code size is calculated by the size of text sections of the binary file. Most modern CPUs have performance counters that can be used to track the cycle number. Thus, this allows a minimal interference calculation of performance (measured in clock cycles) of applications running on them. For example, the Intel XScale processor has a special register *CCNT* that counts core clock cycles.

The most difficulty task might be the measurement of power consumption. Currently, there are two approaches to complete this task: physical measurement and simulation.

Generally speaking, objective and precise results could be obtained by measuring the real hardware. However, there are some limitations with physical measurement. First, the hardware must be ready before conducting such measurement. Second, for most systems, it is nearly impossible to measure different system components separately. Third, without statistics of the execution of the program, such as the number of instructions executed, the number of memory access operations, etc, physical measurement results are often not able to explain the observed power behavior.

On the other hand, simulations are often used to test architectural ideas and assess system performance and power consumption. Simulators provide the flexibility to modify and analyze the impact of various architectural parameters and components as well as enable more detailed statistics collection than physical measurements on hardware. In power/energy simulations, the system is often modeled as an ensemble of its sub-components. The power consumption of the executing program is estimated as the sum of the energy consumption of all of its (simulated) subcomponents. According to the level of granularity of the simulation model, most of these simulators can be classified as instruction-level, and microarchitecture-level simulators.

However, simulations also have their own problems. First, in order to design and implement the simulator, developers need to have detailed knowledge of the internal microarchitecture of the processor. Sometimes, such knowledge is not available to researchers outside the hardware manufacturer. Second, to obtain high precision data, simulators have to model a very detailed version of the architecture and thus the simulations could be very slow. Moreover, due to various sources of error[8], there may be considerable mismatches between the hardware platform and the simulation.

The following two instances shows briefly the main considerations of configuring the evaluation platform in practice.

3.3. Instances of Evaluation Platforms

- **Platform on XScale**
  This platform uses the technique of physical measurement. It consists of the KCC toolchain and the Intel XScale architecture. The corresponding physical measuremental platform for Intel XScale 80200 Evaluation Board (80200EVB) was developed. With hardware support and run-time libraries, both the performance and the power consumption were accurately measured. The main components of this kind of plat-

form are the hardware testbed, a specific developed system library on this testbed and a series of utilities for measuring operations.

- **Platform on CELL**
  This platform consists of the IBM full-system simulator, MAMBO, for the Cell architecture and the GNU toolchain for such architecture. This toolchain has two different compilers designed to produce code for Cell's synergistic processor element (SPE) and Cell's PowerPC processing Element (PPE). A brief flowchart of the platform is shown in Figure 2.

  This platform is a typical instance of measurement using simulation. The underlying simulator should support both cycle-accurate and power simulation. Usually, an extension of the current simulator, or a new simulator, are needed to satisfy these purposes.

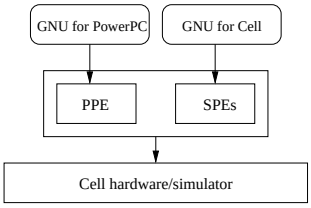

Figure 2. Cell Platform

4. Narrowing Down the Combinational Space

4.1. A Description of the Combinational Space

Most of modern compilers have a large number of options to control various aspect and degrees of optimization. The number of possible combinations of these options is astronomical. Because a majority of compiler users have no in-depth knowledge about what a given option exactly does and how such option interact with others, they usually use the standard compiler switch -*O*x(x =1,2,3). However, the underlying compiler is pre-designed for a specific architecture. Therefore there is no custom combination of compiler options that can be used by all applications. Intuitively, there should be an optimal combination of options for each application. Finding such optimal combination of options for a specific application is almost impossible in practice. An alternative way is to find an optimized combination of options. A combination of options is optimized if:

using this combination of options, the compiler is able to generate "better" code than using the standard compiler -O*x switch*.

Here "better" refers to, when compare with the standard compiler switches, either to faster execution time, lower power consumption, smaller code size, or a combination of them, depending on what is the most important factor for the user.

It is obvious that there is more than one optimized combination of options. The main objective here is to narrow down the combinational space into a smaller space(Ω*_space*) which contains only the optimized combinations for a specific application domain. From now on, we will use *C_space* to refer the space of all possible combinations of compiler options.

4.2. Methods of Finding Ω*_space*

Several methods of finding the Ω*_space* from *C_space* have been proposed in the literature. Two of these methods are of special interests to us: the method employing a genetic algorithm[1, 2, 3, 4] and the method using a statistical technique[5, 7, 10].

4.2.1. Method via Genetic Algorithm

A genetic algorithm (GA) is an iterative process based loosely on biological evolution through natural selection. A particular efficacy of GA is its ability to search large solution spaces.

The method using GA to find the Ω*_space* is presented in the following steps:

- The initial population is built from a random combinations of compiler options (organisms);
- In each generation, measure the benchmark packages for each combination of options. The measurement results(execution time, power or code size) from each combination are set as its "fitness". Smaller fitnesses are "more fit" , and more likely to reproduce; mutation and migration between populations introduces variations to prevent populations from stagnating.
- Using GA to cycle through the generations and refine the best combination of options through natural selection, options that produce optimized result will occur more often, while adverse options will tend to be winnowed away.
- Count the number of times that an option is enabled by the best chromosome in each generation of each population. The higher the count, the more often the option was enabled, and the more important it is for producing optimized results on the given benchmarks. Conversely, an option that is detrimental will appear very few times (or not at all), while neutral options (those that have no effect on the result) should occur an average number of times.
- The optimistic and pessimistic options are then determined using some strategies. For example, [1] takes the final totals and calculate a *z score* for each option.

The *z score* measures the distance of a value from a population's average, in units of standard deviation. If an option's *z score* is greater than 1, it may be beneficial, while a *z score* of -1 or less indicates a detrimental option.

4.2.2. Method via Statistical Technique

This method is used to find the Ω*_space* by really measuring the benchmarks using the chosen combinations of options. It is based on a technique called fractional factorial design (FFD). It can be used to reduce the number of evaluation runs significantly. As described in [5], in the terminology of evaluation design, each compiler option is a factor. Each subset of the factors is a combination of compiler options. The number of the factorial designs consist of the *C_space*. To create a Ω*_space*, FFD systematically selects a subset of the evaluation runs from the factorial design(*C_space*). Some ambiguities may arise when the number of runs is reduced. However, the ambiguity can be resolved in some later evaluation. If the uncertain interactions have significant performance impact, evaluations have to be further refined to resolve the ambiguities. In this way, the total number of runs(Ω*_space*) will be significantly less than a full factorial design(*C_space*).

4.3. Create the Ω*_space* in *UMECO*

The method that creates the Ω*_space* in *UMECO* employs both the genetic algorithm and the statistical technique described above(A detailed description of how to use the genetic algorithm and the statistical technique in UMECO to create the Ω*_space* is not given in this paper).

In contrast to the previous studies which only considered one specific factor, our approach integrates three primary factors: performance, power and code size.

We introduce a 3-tuple of the form <execution time, power, code size> to store the results of the three factors corresponding to one combination of compiler options. The process of creating the Ω*_space* is described as following:

- Setup Baselines
  Baselines represent the best results of performance, power or code size for a given measurement package when the standard compiler switch -*O*x is used. There is no unique compiler standard switch that is universal suitable for all kind of application packages and architectures. The choice of baseline relies on the underlying compiler and architecture pair.
- Initialize the Ω*_space*
  Each factor (performance, power and code size) corresponds to a specific Ω*_space*, we name them as Ω*_space*(T), Ω*_space*(P) and Ω*_space*(S) (where T is performance, P is power and S is code size). The initial elements in these spaces are the standard switch that is used to set the baseline and all individual options.
- Extend the Ω*_space*
  Extend the Ω*_space* to its final status is an iterative process:
  - Measure each option (except the baseline switch) in the Ω*_space*. Repeatedly remove the option that has the worst result from the Ω*_space* until *n* options are left. Here *n* is assigned in the real measurement environment, such as 10 or 15.
  - Use GA algorithm and statistical technique to reduce the number of measurement among the 2^n possibilities of combinations of options.
  - Repeat above step for all combinations that have at least two options and at most all options in Ω*_space*. Each iteration generate a new generation of Ω*_space*. Each generation always contains *n* elements which has the best measurement results.

Each element in the Ω*_space*(f) (f=T,P,S) and the corresponding measurement results compose an entry in the specific database(f). The elements in the database are stored in increasing-order according to the value of the specific factor (performance, power and code size). For example, the first element in a database(P) has the lowest of power consumption among all other elements in that database.

5. Benchmark Packages

There are two kind of benchmark packages in *UMECO*: benchmark packages for testing and benchmark packages for measuring. When there is more than one application domain supported by a specific compiler-architecture pair, a set of measuring benchmark packages and a set of testing benchmark packages should be chosen for this compiler-architecture pair.

5.1. Benchmark for Measurement

This kind of benchmark packages is used to create the databases of optimized combinations of compiler options for the application domains that the compiler-architecture pair support. There are commonly recognized benchmark packages for each specific application domain. For example, consider the benchmark choice for the GCC compiler and XScale architecture pair. One of the embedded applications that the pair supports is the multimedia domain. We choose Mediabench as the underlying measuring package and Mibench as the corresponding test benchmark in the initial phase of creating the databases.

5.2. Benchmark for Testing

This kind of benchmark packages is used to test the effectiveness of the revised compiler. Although some common benchmarks can be used either as for measuring or for testing in *UMECO*, only a real application can be used to confirm the practical value of our methodology.

5.3. A Glance of Measurement and Testing

Using benchmarks to measure and test can be an iterative refined process. In the first step, choose benchmark $\boldsymbol{B}_1$ for measuring. It produces three databases for performance, power and code size, respectively. These databases are integrated in the compiler. Then, choose benchmark $\boldsymbol{B'}_1$ for testing to confirm that optimized results can be obtained when the revised compiler is used. In the next iteration, the benchmark $\boldsymbol{B'}_1$, that was used in the previous iteration for testing can be used for measurement. The new generated databases can be merged with the previous databases. A new benchmark $\boldsymbol{B}_2$ can be selected for testing. This iterative refined process can be repeated many times as needed.

6. Compiler Revision

An obvious drawback of most modern compilers is the lack of user interferences. It is impossible for the underlying compiler to know which domain an application belongs to, and, thus, what would be the optimal combination of optimization options for this specific application.

The *UMECO* uses a specific compiler-user interface for the users to interface with the compiler. Then, the compiler has a revision phase to accept the advice information from the users and to choose a combination of options based on the advice information and a pre-built databases.

In this section, we present the syntax of the user-advised options, and the internal flowchart of the compiler to process the user advice options.

6.1. Syntax of User-Advised Options

We add two new options to the underlying compiler command line: the *priority-weight* option and the *application-domain* option. The syntax and the meaning of the options are described as follows:

- Priority-weight option

  $$\boldsymbol{-PW=< p_1(w_1), p_2(w_2), p_3(w_3) >}$$

  This option is used to assign priorities to performance, power and code size. The p_i(i=1,2,3) parameter can be either the character ***P*** (for performance), ***W*** (for power) and ***S*** (for code size). The users use this option to advise the underlying compiler what is the most important factor about the application that is being compiled. For example, $-PW=<\mathbf{W}>$ means that the user only cares about minimizing the power consumption.

  The w_i(i=1,2,3) parameter is a weighted coefficient and its value is a fraction between $[0,1]$. The sum of w_1, w_2 and w_3 must ≤ 1. Users can use this option to advise the compiler the importance, and possible trade off, for each factor (performance, power and code size). The compiler uses these weighted coefficients in its trade-off algorithms(not described in this paper).

- Application-domain option

  –D=class

  This option tells the compiler what domain of the application to be compiled belongs to. The ***class*** parameter is a character of can be an ***N*** (network application domain), a ***D***(DSP application domain), a ***W***(wireless communication domain), a ***M***(media application domain), etc.

6.2. Compiler Modification

Basically, the revision to the underlying compilers includes modifications to extend the compiler command line with new options, integrate the databases that are created in the measurement phase, modify the internal driver of the compiler to accept the user-advised options and implement new algorithms that provides a trade off among performance, power and code size, as shown in Figure 3.

```
1  Read Command Parameters
2. if has "-PW" option then
3. {
4.    if has "-D" option then
5.      if has weighted parameter then
6.        optimization_list = trade_off_function (weighted_coefficient_list, class);
7.      else { /* no weighted parameter */
8.        search the database corresponding to the -PW option in class D;
9.        optimization_list = the first item of the searched database;
10.     }
11.   else { /* no -D parameter */ {
12.       search the database corresponding to the -PW option in the common class;
13.       optimization_list = the first item of the searched database;
14.   }
15. else /* no -PW option */
16.   enter to the original process;
17. }
```

Figure 3. Compiler Revision

Suppose that a domain option ***–D*** is given, together with a no weighted ***–PW*** option, which means the user is only

concerned about one factor (either the performance, power or code size) for this specific application domain, the underlying compiler simply chooses the combination of options of the first position from the database that corresponds to the specific application domain, as the elements in the database are stored in increase-order according to the value of the specific factor.

If the user requires a trade-off among performance, power and code size using the weighted coefficients, the compiler needs to search all the databases that correspond to the specific application domain and finds the suitable combination of options using the appropriate algorithms.

7. Related work

There are few research areas that explores the compiler optimization options to improve performance, lower power consumption, or reduce code size.

A current study of optimal usage of compiler options to improve the performance in applications is reported in [7]. It uses a statistical technique to generate a search space of compiler options and trim down the search space by an orthogonal array. Although the experimental results show the efficiency of this approach, we are not aware that automatic compiler support exists for this approach. From the view of the application developers, they would be responsible to search for the optimal combination of compiler options since no compiler support is provided.

Other studies of finding an optimal program specific compilation sequence is studied in literature [6, 11]. Basically, the idea in these studies is to hack compilers to adaptively adjust their behavior to produce the best code that they can in any particular circumstance. However, this approach requires an extensive understanding of compilation optimization techniques. Also, these studies are in the beginning stage.

Most of the above studies only consider one specific factor, especially performance, or discuss the general issues of why and how a combination of compiler optimization options affect performance improvement or power dissipation. However, our approach integrates the three primary factors of performance, power and code size as a whole and gives an integrated methodology of automatic exploring compiler options for different application domains.

8. Conclusion

The traditional selection methodology of optimization options is one of the main obstacles for users to fully utilize the compiler optimizations. In this paper, we proposed a user-advised methodology for automatic exploration of compiler optimization options.

The methodology proposed in this paper clearly can be applied to a vast number of computational and embedded systems. We are now implementing this methodology on three well known embedded and multicore architectures: Intel XScale, IBM Cyclops-64 and Cell. To confirm the effectiveness of this methodology, tests with real applications are necessary. We are still looking forward to external cooperation with real users. This will extend our methodology to more compiler-architecture pairs. For revision purposes, the compilers should be open source. Fortunately, most commercial compilers are based on GNU compiler which opens its source to the public.

Acknowledgments

We wish to acknowledge our sponsors from DOD, DOE (Award No. DE-FC02-01ER25503) , and NSF (Award No. CCF-0541002 and CNS-0509332).

References

[1] *S. R. Ladd, Describing the Evolutionary Algorithm*. http://www.coyotegulch.com/products/acovea/acoveaga.html.

[2] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, May 1999.

[3] T. Kisuki, E Knijnenburg, and M. O'Boyle. *Combined selection of tile sizes and unroll factors using iterative compilation*. 2000.

[4] A. Nisbet. *Genetic algorithm optimized parallelization*. 1998.

[5] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *2nd ACM Workshop on Feedback-Directed Optimization (FDO)*, Haifa, Israel, November 1999.

[6] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *LCTES*, pages 231–239, 2004.

[7] M. Haneda, P.M.W. Knijnenburg and H.A.G. Wijshoff. Optimizing general purpose compiler optimization. In *CF'05*, Ishia, Italy, May 2005.

[8] D. B. R. Desikan and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266 – 277, June 2001.

[9] Richard Stallman. *Using and Porting the GNU Compiler collection (GCC)*. Free Software Foundation, Inc., 2000.

[10] R.P.J. Pinkers, P.M.W. Knijnenburg, M. Haneda and H.A.G. Wijshoff. Analysis of compiler options using orthogonal array. In *Proceeding of CPC*, 2004.

[11] N. V. S. Triantafyllis, M. Vachharajani and D. August. Compiler optimization-space exploration. In *Proceedings of the '03 International Symposium on Code Generation and Optimization*, pages 204–215, 2003.