On the Impact of Data Input Sets on Statistical Compiler Tuning

M. Haneda[1], P.M.W. Knijnenburg[1,2], and H.A.G. Wijshoff[1]

[1]LIACS, Leiden University
The Netherlands
{haneda,peterk,harryw}@liacs.nl

[2]University of Amsterdam
The Netherlands
peterk@science.uva.nl

Abstract

In recent years, several approaches have been proposed to use profile information in compiler optimization. This profile information can be used at the source level to guide loop transformations as well as in the backend to guide low level optimizations. At the same time, profile guided library generators have been proposed also, like Atlas, Spiral, or FFTW, that tune their routines for the underlying hardware. These approaches have led to excellent performance improvements. However, a possible drawback of these approaches is that applications are optimized using a single or a limited set of data inputs. It is well known that programs can exhibit vastly differing behaviors for different inputs. Therefore, it is not clear whether the performance numbers reported are still valid for other input than the input used to optimize the program. In this paper, we address this problem for a specific statistical compiler tuning method. We use three different platforms and several SPECint2000 benchmarks. We show that when we tune the compiler using train data, we obtain a compiler setting that still performs well for reference data. These results suggest that profile guided optimization may be more stable than is sometimes believed and that a limited number of train data sets is sufficient to obtain a well optimized program for all inputs.

1 Introduction

Code optimization at compile time has been a main focus of compiler research since the advent of the first compiler. Many optimizations have been proposed, both in the backend such as common subexpression elimination or strength reduction [1], as well as in the front-end where code restructuring can be performed, such as loop unrolling, loop tiling, or loop interchange [16]. It is well known that the best optimization sequence depends on both the application as well as the target architecture. Therefore, application developers usually spend much time to carefully hand tune an application.

In recent years, several approaches have been proposed to automatically find optimizations that are beneficial for an application by exploiting profile information [3, 13, 6, 17, 15, 5, 14, 18, 10]. At the same time, several library generators have been developed, like Atlas [23], Spiral [19], and FFTW [7]. These generators use small kernels to probe the underlying hardware and profile information to highly tune BLAS and DSP routines. All these efforts report performance of applications and routines that are better than the performance obtained from standard compilers and hand-optimized, vendor supplied library routines. Therefore, the incorporation of profile and feedback directed optimization engines in a compiler seems to be a promising direction to obtain highly optimized codes.

However, a possible drawback of all profile based approaches to program optimization is that only a single or at most a limited number of data input sets are used to obtain runtime information about the application to be optimized. Moreover, performance numbers reported in published papers usually come from the same inputs as used in the optimization search. However, it is well-known that an application can exhibit widely different behavior for different inputs. Therefore, it is not immediately clear that a sequence of optimizations found for a particular input is still a good one for a different input. Different dynamic execution paths may well require different optimizations. However, the converse may also be true: for a given application, most execution paths benefit from more or less the same optimizations. For example, in `mpeg2enc`, a heavily used routine is the motion compensated prediction routine. This routine has eight different loops, one of which is selected depending on the value of the `addflag` variable and other parameters. However, all these loops have the same structure and they will therefore benefit from the same optimizations.

In this paper, we have experimented with a statistical compiler tuning methodology, described earlier in [10]. We have optimized several SPECint2000 programs using their train data sets. Next, we have ran the optimized programs on their reference data. We show that the performance they obtain is in line with their performance on train data for all programs on three different platforms. These results suggest that the optimization obtained from using profiles from one data input set is actually more stable across inputs than is sometimes feared.

This paper is structured as follows. Section 2 explains inferential statistics and the Mann-Whitney test briefly. Our iterative algorithm is given in Section 2.4. Section 3 describes our experimental environment and we discuss our results in Section 4. Related work is discussed in Section 5 and we summarize the paper in Section 6.

2 Background: Statistical Compiler Tuning

In this section, we briefly discuss our statistical compiler tuning methodology, first proposed in [10].

2.1 Inferential Statistics

Most experiments in the scientific field aim to support a prediction or an estimation of a phenomenon. The prediction or estimation is called an *experimental hypothesis*, and the study of inferential statistics aims to predict whether an experimental hypothesis is likely to be true. In the present paper, the experimental hypothesis reads

Experimental Hypothesis Compiler option A is effective to optimize application B.

The following three steps describe the basic idea of inferential statistics. First, we define a *null hypothesis* which negates the experimental hypothesis. When we want to know about the effectiveness of compiler option A for application B, the null hypothesis is

Null Hypothesis Compiler option A is *not* effective to optimize application B.

Second, we conduct an experiment which contains two groups which are called the *control group* and the *experimental group*, respectively. The control group consists of the experimental runs that do not use compiler option A. The experimental group consists of the experimental runs which use compiler option A. The null hypothesis implies that the execution times from these two groups are the same. Hence, if we can conclude that the execution times from these two groups are significantly different, then we may reject the null hypothesis and accept the experimental hypothesis. We discuss our experimental groups is Section 2.2.

Hence, thirdly, we need a method to assess the difference between two groups. Inferential statistics provides a so-called test statistic to evaluate this difference. The test statistic enables us to assess a confidence rate to support an experimental hypothesis. We discuss our test in Section 2.3.

2.2 Experimental groups

In order to test a hypothesis, we need to define a control group and an experimental group. We could define these groups for each option to be tested. However, this would entail very many experiments we would need to perform. Therefore, we would like to create one experimental group and one control group that can be used to test all options. This means we would need to construct a collection of more or less random settings. For each option, we take for the experimental group those settings that turn the option on and for the control group those settings that turn it off. In this way, the profile information obtained for one particular setting will be used to test each option, either as part of the experimental group or of the control group. In order to achieve this, we use a so called *Orthogonal Array* [11].

Briefly, an Orthogonal Array (OA) is a matrix of zeroes and ones. The rows of an orthogonal array represent experiments to be performed and the columns of the orthogonal array correspond to the different factors whose effects are being analyzed. For the purposes of this paper, an OA has the property that for two arbitrary columns, the patterns

$$0\,0 \qquad 0\,1 \qquad 1\,0 \qquad 1\,1$$

occur equally often. For example, the left part of Table 1 shows an Orthogonal Array.

An OA has the property that any option is turned on and off equally often in the experiments defined by the rows of the OA. Moreover, for the rows that turn a certain option on, any other option is turned on and off equally often as well. This means that for each factor, there exist $\frac{1}{2}N$ rows that turn this factor on, and $\frac{1}{2}N$ rows that turn this factor off, when N is the total number of rows. Hence, by using one Orthogonal Array, we can perform measurements that can be used in the Mann-Whitney test for every option. At the same time, the other factors have values that cover the entire space reasonably. Thus, we can measure the impact of an option in an arbitrary context. The Orthogonal Arrays used in the present paper are taken from [20].

2.3 The Mann-Whitney Test

In this section, we discuss the statistical test we employ. Since both experimental and control group consist of experiments where many different options are turned on, the execution times of the members of each group can differ considerably. Therefore, it is not valid to just take the average execution times for each group and compare these averages. Because of the large variation in each group, a difference between these averages could well be by pure chance. *Non-parametric statistics* are designed to deal with this situation [12]. It is capable of analyzing data without

O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	Time	Rank
0	0	0	0	0	0	0	0	0	0	20	8
1	0	1	0	0	0	1	1	1	0	25	12
1	1	0	1	0	0	0	1	1	1	15	3
0	1	1	0	1	0	0	0	1	1	17	5
1	0	1	1	0	1	0	0	0	1	18	6
1	1	0	1	1	0	1	0	0	0	14	2
1	1	1	0	1	1	0	1	0	0	23	11
0	1	1	1	0	1	1	0	1	0	13	1
0	0	1	1	1	0	1	1	0	1	19	7
0	0	0	1	1	1	0	1	1	0	22	10
1	0	0	0	1	1	1	0	1	1	21	9
0	1	0	0	0	1	1	1	0	1	16	4

Table 1. Example of Experimental Settings for 10 Compiler Options

	Experimental group ($O_2 = 1$)	Control group ($O_2 = 0$)
	3	8
	5	12
	2	6
	11	7
	1	10
	4	9
Total	$T_1 = 26$	$T_2 = 52$

Table 2. The Effect of Compiler Option O_2

an underlying distribution by ranking the raw data first and analyzing the rankings.

The Mann-Whitney test is a well known test in inferential non-parametric statistics [12]. We explain the Mann-Whitney test using an example. Table 1 shows an experimental design and the resulting execution times for a compiler with ten options. The first ten columns correspond to the ten compiler options and each row expresses a total compiler setting. The 11th column shows the execution times using these settings. The last column shows the ranks of the execution times in ascending order. That is, the fastest setting is assigned rank 1, the second fastest setting rank 2, etc.

Suppose we want to decide whether compiler option O_2 affects the optimization process or not. The experimental data is separated into the experimental and control group. The ranks for these groups are shown in Table 2. The last row of the table is the sum of the ranks for each group, denoted by T_1 and T_2, respectively. We see that T_1 and T_2 are different, but are they different enough to conclude that option O_2 has a significant effect?

The Mann-Whitney test is based on the value of T_1. In order to discuss how the test works, assume for simplicity that the two groups both contain N members and that the option to be analyzed is the only difference between the groups. Suppose further that option O_2 is not effective, that is, that the null hypothesis is true. Then the assignment of ranks is basically random, resulting from experimental errors in the measurements. Looking at which values T_1 can have, T_1 is at least $1 + 2 + \cdots + N$ and at most $(N+1) + \cdots + (2N)$. The first value occurs when all measurements in the experimental group happen to be slightly smaller than the measurements from the control group. The second value occurs in the opposite case. For an intermediate value, there are more possibilities to rank the measurements and obtain this value. Hence, the chance that T_1 has such an intermediate value is larger. It has been shown [12] that if the null hypothesis is true, then T_1 has a *normal distribution* with mean

$$\mu = \frac{N(2N+1)}{2} \tag{1}$$

and standard deviation

$$\sigma = \sqrt{\frac{N^2(2N+1)}{12}} \tag{2}$$

Since T_1 is normally distributed, we can apply 'ordinary' statistics on it.

The Mann-Whitney test does not consider T_1 directly but considers the test statistic z instead, which is given by

$$z = \frac{T_1 - \mu}{\sigma} \tag{3}$$

That is, z measures how far T_1 lies from the mean expressed in units of standard deviation. Then z is normally distributed also (with mean zero) and this distribution is given by

$$Y(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}z^2} \tag{4}$$

The normal distribution expresses the chance to measure a certain value for z. Hence,

$$\int_{-\infty}^{\infty} Y(z)dz = 1$$

If the measured value of T_1 is significantly different from μ, then we may conclude that the null hypothesis is false because it is highly unlikely that we measure such a value by chance. In order to decide whether T_1 is significantly different from μ or, equivalently, whether its corresponding value z is significantly different from zero, we proceed as follows. Consider the function $P(t)$ given by

$$P(t) = \left(1 - 2 \cdot \int_0^t Y(z)dz\right) \cdot 100\% \tag{5}$$

Then $P(t)$ expresses the chance to measure a value for z such that either $z \geq t$ or $z \leq -t$. A standard criterion [12] for "significant difference" is when the chance to measure a certain value of z is less than 5%. This threshold of 5% is called the *critical value* of the test. The corresponding value for t is 1.96, as indicated in the figure. This means that the chance of measuring a value for z that is larger than 1.96 or smaller than -1.96 when the null hypothesis is true, is less than 5%. This essentially means that the probability to reject the null hypothesis when it is in fact true, is less than 5%.

To sum up, the Mann-Whitney test calculates the rank sum of the experimental group and the corresponding value for z. Then it calculates the value $P(|z|)$ and checks whether this is smaller than 5%[1]. If this is the case, we conclude that the experimental group is significantly different from the control group and the null hypothesis is rejected. Note that in this test, the control group is only used to derive ranks for the experimental group.

In the current example, $N = 6$ and $T_1 = 26$. Hence, the z value for O_2 is computed as follows:

$$\begin{aligned} \sigma &= \sqrt{\tfrac{6\cdot 6\cdot(12+1)}{12}} = \sqrt{39} \\ \mu &= \tfrac{6\cdot(1+12)}{2} = 39 \\ z &= \tfrac{(26-39)}{\sqrt{39}} = -2.08 \end{aligned}$$

By using Equation (5), we determine whether the observed data satisfies the criterion $P(|z|) < 5\%$. Since $z = -2.08$, this yields $P(|-2.08|) = 3.75\%$. Hence O_2 satisfies $P(|z|) < 5\%$, and we can conclude that compiler option O_2 has a significant effect. The obtained z value is negative so that the observed data in group 1 implies shorter execution times than group 2. This means that the effect of O_2 is positive. This procedure can be applied to each optimization in Table 1. Therefore, we can determine the important options among the 10 options with 12 experimental runs.

2.4 Statistical compiler tuning

This section describes our algorithm to determine a compiler setting for an application based on the statistical theory discussed in the previous section.

The algorithm tries to detect compiler options with a significant effect. It starts with a factor list which includes all compiler options, and produces a compiler setting using an appropriate Orthogonal Array. We use execution times to obtain the test statistic for each compiler option, and this tells us which options have a significant effect, and whether they should be turned on or off. The compiler options whose settings are determined are removed from the factor list, and the reduced factor list is used to design the next experiment in which a smaller Orthogonal Array can be used. The algorithm starts to explore a large search space in which there is much variation, and it cuts down the search space every iteration, obtaining new search spaces with less variation.

- Choose an orthogonal array A with as many columns as there are options.
- Repeat
  - Compile the application with each row from A as compiler setting and execute the optimized application.
  - Compute test statistic z for each compiler option with equation (3).
  - If the test statistic meets $P(|z|) < 5\%$,
    - If z is negative then the compiler option has a positive effect, and the option is turned on.
    - If z is positive then the compiler option has a negative effect, and the option is turned off.
  - Remove the compiler options that have been selected from the factor list and drop the corresponding columns from A.
- Until
  - All options are set, or
  - No option with a significant effect is detected anymore, or
  - The experimental data has not enough variation (low standard deviation) to apply the Mann-Whitney test meaningfully.
- The resulting compiler setting is the setting obtained by setting all remaining options that have not been selected yet to *off*.

Please note that the final setting obtained by this algorithm differs slightly from the one described in [10]: in that paper we take the setting that produces the best result in the last iteration instead of setting remaining options to *off*. Although [10] obtains slightly better results than the present method, we feel that our present choice is more focused on the power of our statistical selection method, without introducing option settings which 'accidentally' happen to occur in the last iteration.

3 Experimental Environment

We use the gcc compiler version 3.3.1, and we use 42 options for our factor list [8], shown in Table 3(a). We did not employ a number of options that optimize floating point operations, like `fast-math` or

[1]Equivalently, we can check whether $|z| > 1.96$ but the above formulation is more intuitive. There exists a simple algorithm to compute $P(|z|)$ from z given in [12].

1	defer-pop	2	force-mem
3	force-addr	4	inline-functions
5	optimize-sibling-calls	6	merge-constants
7	strength-reduce	8	thread-jumps
9	cse-follow-jumps	10	cse-skip-blocks
11	rerun-cse-after-loop	12	rerun-loop-opt
13	gcse	14	loop-optimize
15	crossjumping	16	if-conversion
17	if-conversion2	18	delete-null-pointer-checks
19	expensive-optimizations	20	optimize-register-move
21	schedule-insns	22	sched-interblock
23	sched-spec	24	schedule-insns2
25	sched-spec-load	26	sched-spec-load-dangerous
27	caller-saves	28	move-all-movables
29	reduce-all-givs	30	peephole
31	reorder-blocks	32	reorder-functions
33	strict-aliasing	34	align-functions
35	align-labels	36	align-loops
37	align-jumps	38	rename-registers
39	cprop-registers	40	function-sections
41	data-sections	42	unroll-loops

(a) Compiler options

Name (#lines)	Description
164.gzip (4333)	gzip (GNU zip) is a data compression program. gzip uses Lempel-Ziv coding (LZ77) as its compression algorithm.
175.vpr (8899)	VPR is a placement and routing program for technology-mapped circuit.
181.mcf (1120)	The program is designed for the solution of single-depot vehicle scheduling problems occurring in the planning process of public transportation companies.
197.parser (6839)	The Link Grammar Parser is a syntactic parser of English.
254.gap (27523)	It implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).
255.vortex (31128)	Vortex is a single-user object-oriented database transaction benchmark.
256.bzip2 (2955)	256.bzip2 is a data compression program.

(b) Benchmarks

Table 3. Experimental Framework: Option List and Benchmark Programs

`unsafe-math-optimizations` since these options violate IEEE and ISO specifications for mathematical functions. The option `delayed-branch` is neither present since one of our target architectures, the P4, does not support delayed branching. Neither did we include several options that are marked as 'experimental' in the manual.

We use 7 benchmarks from the *SPECint2000* benchmark suite, shown in Table 3(b). We used the train data set as the input for the benchmarks in order to reduce profiling time.

We use the following target architectures for our experiments: a Pentium 4 2.8 GHz processor, a SPARC Sun Fire V25 server dual 1.28 GHz processor and a IA64 dual Itanium2 1.296 GHz processor.

We have used the `time` command to measure execution times. Execution time can be different when we run the same executable several times. To smooth out this experimental fluctuation, we run the same executable code 10 times, and take the average value of these runs. However, if a run causes the standard deviation of all 10 runs to change by more than 0.5% of the average value, then this run is deleted from the experiments, and the overall average is taken from the remaining runs.

Below, improvements are calculated as follows. First, any `-Ox` switch also 'silently' turns on some other optimizations than listed in the manual, in particular, register allocation. We define the O_{base} setting as the setting resulting from switching on `-O` and explicitly turning off all options. Then only the hidden options are active. The improvement I_P of our new setting O_{new42} for an application P is given by

$$I_P = \frac{T_P(O_{base}) - T_P(O_{new42})}{T_P(O_{base})} \cdot 100\%$$

where $T_P(O_x)$ denotes the execution time of application P when compiled with compiler setting O_x.

We have used an Orthogonal Array with 42 columns (corresponding to the 42 compiler options) and 48 rows (giving us a *power* of 80%, as discussed in [10]), which is borrowed from [20].

4 Results

In this section, we show the results we obtained using our methodology on three platforms when we select compiler options using train data. Next, we show the improvements we obtain when we run the resulting optimized program on reference data.

In Figures 1 through 3 we show the improvements for `-O3` and the setting obtained from the Mann-Whithney test, denoted by O_{new42}, for three different platforms. We have used 7 SPEC2000 benchmarks for the P4 and 5 for the IA64 and SPARC since the two other benchmarks did not compile correctly on these platforms. We have used the train data to run our methodology and we show improvements when using train data again. The numbers in brackets after the benchmark name denote the number of iterations required to fully run the selection method. In this paper, we have let the method run until no more option could be selected, usually due to the fact that the measured execution times from one iteration did not have sufficient variation to apply the Mann-Whitney test meaningfully. Please note that this number of iterations can be reduced significantly by setting the threshold value for this variation higher, as we have shown in [10], without reducing the improvements obtained.

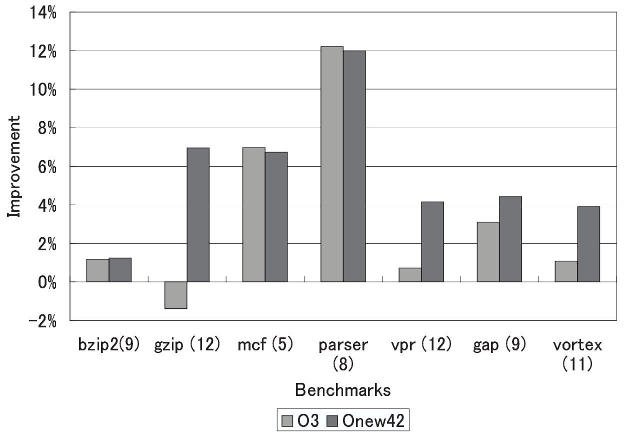

Figure 1. Improvements of `-O3` **and** O_{new42} **for P4 using train data**

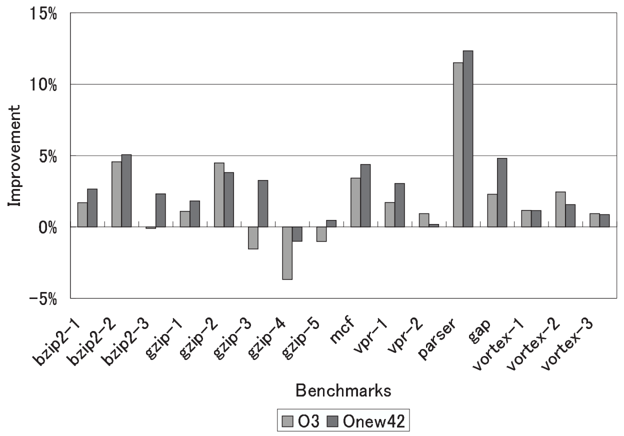

Figure 4. Improvements of `-O3` **and** O_{new42} **for P4 using reference data**

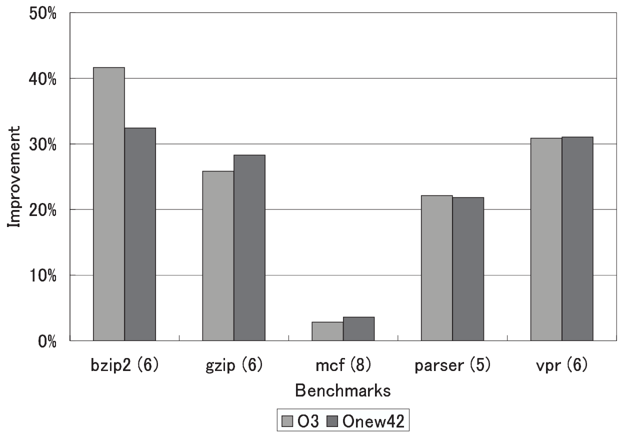

Figure 2. Improvements of `-O3` **and** O_{new42} **for IA64 using train data**

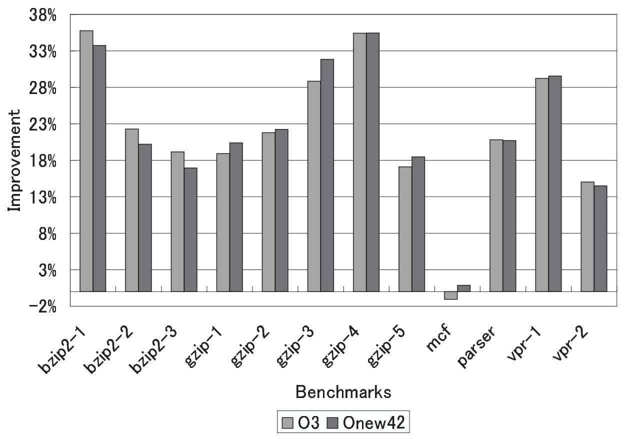

Figure 5. Improvements of `-O3` **and** O_{new42} **for IA64 using reference data**

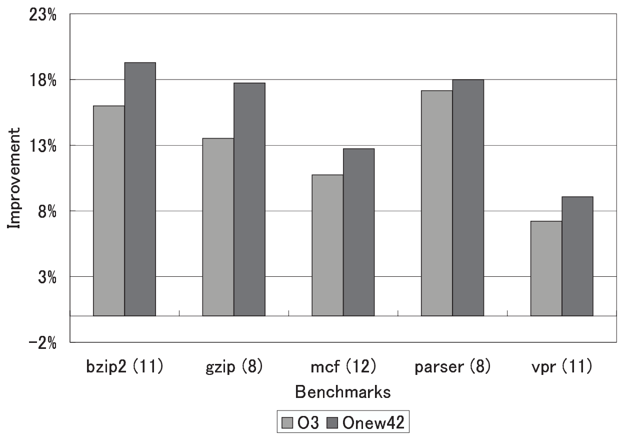

Figure 3. Improvements of `-O3` **and** O_{new42} **for SPARC using train data**

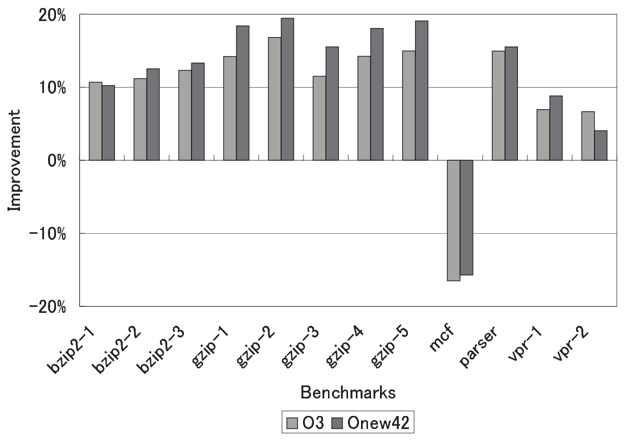

Figure 6. Improvements of `-O3` **and** O_{new42} **for SPARC using reference data**

We immediately observe that for the P4 and the SPARC, significantly better improvements are obtained than `-O3`. The situation is different for the IA64 where our method actually reduces the improvements for `bzip2`. For the other benchmarks, the improvements that we found are almost equal to the improvements obtained by `-O3`. A possible explanation for this last observation is that almost all improvement comes from correct instruction scheduling for this EPIC architecture and that little extra improvement can be found using other options.

In Figures 4 through 6 we show the improvements when we run our optimized program on the reference data. Since the benchmarks have several distinct reference data sets, that should be used together, we show the results when running the optimized programs on all reference data sets. We observe that for the P4 and the SPARC, generally better performance is obtained using our new setting. For the IA64, the improvements we obtain are more or less equal to the improvements of `-O3`, much in line with the previous results. Comparing these figures with the previous ones, we immediately observe that the performance improvements are much in par. In fact, for the P4, the improvements for `mcf` and `parser` are actually better than for the train data. However, for `vortex`, the situation is reverse. For the IA64, the improvements for `bzip2` for the train data are also less than for the reference data. Finally, on the SPARC, `bzip2` improvements are slightly better for train data than for reference data. However, for the vast majority of our benchmarks, performance improvements for the train data (which is used to select the compiler options) is more or less the same as for the reference data.

We conclude that the dependence of the SPECint2000 benchmarks on data input is not as strong as is sometimes believed. Profile guided compiler tuning using one (representative) data input set produces compiler settings that work well for other data input sets. This result is of importance to other approaches that use profile guided search to optimize programs or library routines. However, a more thorough investigation of this dependence is in place, in particular for high level loop transformations that may exhibit a stronger dependence. Nevertheless, the results in this paper show that we may be optimistic and that profile guided compiler searches may need only a limited set of representative inputs to produce good results for many or even all inputs.

5 Related Work

Traditional compiler texts [1] do not address the problem of which optimizations to use. [16] presents a static list based on his experience. A number of approaches to select best optimizations have been proposed by searching the optimization space. Iterative compilation [3, 13] searches for source level transformations. [17] uses genetic algorithms to find optimal source level transformations. These approaches, however, seem to require many thousands of program runs. [15] and [21] use machine learning techniques to find compiler heuristics. [22] discusses an implementation of iterative compilation in the Intel IA-64 production compiler. In contrast to these efforts, our approach uses statistical analysis to systematically prune the search space and is focused on compiler switches.

Granston and Holler [9] propose a tool for automatic selection of compiler options, called *Dr. Options*. This tool uses information about the application supplied by the user and a set of tuning rules that have been created by interviewing tuning experts and analyzing optimization experiments. VISTA [26] is an interactive tool to assist the application programmer in finding optimizations and their phase order. [25] provides a framework for predicting the impact of loop transformations to assist in selecting the optimal one. Whitfield and Soffa [24] propose a framework for specifying transformations and an automatic optimizer generator in order to experiment with transformations. However, this approach requires much knowledge about the compiler and does not solve the problem of which transformations to enable automatically.

Cooper et. al. [6] propose to use genetic algorithms to select options using a research compiler paying attention to both code size and speed. However, they only employ 10 options, in contrast to the present approach which uses 23 factors. It is not immediately clear that a large number of options will not lead to combinatorial explosion in their approach. Moreover, their compiler allows them to specify the order in which these optimizations are applied and the same optimization may occur several times in an optimization sequence. In contrast, we use an existing production compiler in which this order is fixed as is generally the case for production compilers. Hence, the technique from [6] is not immediately applicable to production compilers. In [2], it has been shown that the number of profiles required can be reduced significantly when the structure of the search space is taken into consideration. The related VISTA system [14] uses GAs to find sequences of backend transformations using far less profiles than the original system.

Chow and Wu [5] determine which option to set using a linear regression model, based on a fractional factorial design using aliasing [4]. This approach has a number of drawbacks compared to the present work. First, it is well known that many aspects of program execution are nonlinear. Hence, it is not clear that such a linear model may be used and the authors do not give an argument for using it. First, to define the aliasing structure requires an initial selection of options which requires knowledge about the compiler. Second, [5] requires many new experiments to resolve ambiguities. In contrast, we propose a simple analysis to iteratively switch on more and more options, zooming in from options having a large effect to options having less effect.

Pinkers et al. [18] use Orthogonal Arrays to calculate the main effect of a compiler option. Options with a large positive main effect are switched on and those with a large negative effect are switched off. There are a number of drawbacks compared to the present approach. First, the method in [18] has several parameters that are dependent on the compiler used. In contrast, our method only uses well established statistical theory and is therefore independent of the compiler. Next, in [18] it is remarked that options should not interfere and having interfering option as separate factors yields lower performance. In contrast, our method seems to be able to cope with this situation. Finally, our method is robust for small bugs in the compiler: if one setting to be tested fails to produce executable code, we can simply adjust the sizes of the populations. In contrast, the method in [18] collapses in this situation.

6 Conclusion

In this paper, we have used an approach to the problem of selecting a compiler optimization setting using inferential non-parametric statistics, in particular, the Mann-Whitney test first described in [10]. The present paper focuses on the dependence of the options selected on the input data used in the iterative search. We have shown that for compiler tuning this dependence is weak: settings found using train data work well for reference data also on three different platforms and seven SPECint2000 benchmarks. This result suggests that profile guided optimization may use a limited set of data inputs to produce results that are good for many other inputs.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

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

[3] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proc. Workshop on Profi le and Feedback Directed Compilation*, 1998.

[4] G. Box, W. Hunter, and J. Hunter. *Statistics for Eperimenters. An Introduction to Design, Data Analysis, and Model Building*. Wiley and Sons, 1978.

[5] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Proc. 2nd Workshop on Feedback Directed Optimization*, 1999.

[6] K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proc. LCTES*, pages 1–9, 1999.

[7] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005.

[8] GNU Consortium. GCC online documentation. `http://gcc.gnu.org/onlinedocs/`.

[9] E. Granston and A. Holler. Automatic recommendation of compiler options. In *Proc. 4th Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[10] M. Haneda, P. Knijnenburg, and H. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *Proc. PACT*, pages 123–132, 2005.

[11] A. Hedayat, N. Sloane, and J. Stufken. *Orthogonal Arrays: Theory and Applications*. Series in Statistics. Springer Verlag, 1999.

[12] M. Hollander and D. A. Wolfe. *Nonparametric Statistical Methods*. Wiley Series in Probability and Statistics, 1999.

[13] T. Kisuki, P. Knijnenburg, and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proc. PACT*, pages 237–246, 2000.

[14] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proc. PLDI*, pages 171–182, 2004.

[15] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proc. AIMSA*, LNCS 2443, pages 41–50, 2002.

[16] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[17] A. Nisbet. GAPS: Genetic algorithm optimised parallelization. In *Proc. Workshop on Profi le and Feedback Directed Compilation*, 1998.

[18] R. Pinkers, P. Knijnenburg, M. Haneda, and H. Wijshoff. Statistical selection of compiler options. In *Proc. MASCOTS*, pages 494–501, 2004.

[19] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE*, 93(2):232–275, 2005.

[20] N. Sloane. A library of orthogonal arrays. `http://www.research.att.com/~njas/oadir/`.

[21] M. Stephenson, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proc. PLDI*, pages 77–90, 2003.

[22] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proc. CGO*, pages 204–215, 2003.

[23] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[24] D. Whitfield and M. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.

[25] M. Zhao, B. Childers, and M. Soffa. Predicting the impact of optimizations for embedded systems. In *Proc. LCTES*, pages 1–11, 2003.

[26] W. Zhao, B. Cai, D. Whalley, M. Bailey, R. van Engelen, X. Yuan, J. Hiser, J. Davidson, and K. Gallivan. VISTA: A system for interactive code improvement. In *Proc. LCTES*, pages 155–164, 2002.