AN EXPERIMENT IN PROGRAM RESTRUCTURING FOR PERFORMANCE ENHANCEMENT

by

Domenico Ferrari and Edwin Lau

Memorandum No. ERL-M580

14 April 1976

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

AN EXPERIMENT IN PROGRAM RESTRUCTURING FOR PERFORMANCE ENHANCEMENT[†]

Domenico Ferrari and Edwin Lau

Computer Science Division
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California at Berkeley

Abstract

An experiment intended to determine the influence of the input data on the performance improvements resulting from program restructuring is described. This influence is found to be non-significant for the particular program, memory policy, restructuring procedure and inputs considered in the experiment. It is speculated that such a conclusion is likely to have much more general validity.

[†]Research supported by National Science Foundation grant DCR74-18375.

1. Introduction

A technique which has been successfully employed to improve the performance of programs running on a virtual-memory system is the one of program restructuring. This technique consists of modifying the relative positions of various parts of a program, called blocks, in the virtual address space so as to increase the locality of the program. Various approaches to restructuring have been proposed. A particularly successful class of approaches can be described by the following five-step restructuring procedure [1]:

(1) the program to be restructured is partitioned into blocks, whose average size must be substantially smaller than the page size (or, in a segmented machine, than the average segment size);

(2) the program is instrumented and run in order to collect data on its dynamic behavior; for the purposes of restructuring, a complete description of this behavior is contained in the block reference string, the string of block references generated by the program during its execution;

(3) a restructuring algorithm is applied to the block reference string gathered in (2); the algorithm produces a restructuring graph whose nodes correspond to the blocks and whose edges have labels somehow representing the desirability of storing the two blocks they connect in adjacent areas of the virtual address space;

(4) a clustering algorithm is applied to the restructuring graph with the objective of determining those clusters of blocks characterized by the minimum sum of inter-cluster labels, under the constraint that the sum of block sizes in each cluster be less than or equal to the page size;

(5) blocks are assigned to pages according to the results of (4), an appropriate ordering of the pages and blocks is determined, and the program is restructured.

This procedure, with different restructuring algorithms, has been applied

to a variety of programs running under a number of different memory management policies. The performance improvements obtained, in terms of indices like the page fault rate and the mean working set size, have been very encouraging, both in the case of programs written for non-virtual memory environments [2-4] and in the case of programs designed for virtual-memory machines [1]. In particular, tailoring algorithms, which exploit the knowledge of the memory policy under which the program will have to run [1,2,5], have been found superior to other restructuring algorithms [3,4].

An objection which has been raised against the restructuring procedure described above is that the characterization of the program's behavior upon which the procedure is based depends on the input data used in the instrumented execution of the program performed in (2). There is no guarantee that the performance improvements due to the reordering of the blocks dictated by the procedure under a certain set of input data will be preserved under different input data. In fact, the new arrangement might, under some inputs, result in a deterioration of performance with respect to the original non-restructured program. Hatfield and Gerald [3] noted that "fortunately, many commonly used programs are rather insensitive to data". Ferrari [1] published the results of many short experiments in which the performances of a program restructured using two different algorithms, were compared under a variety of inputs. Both for an interactive text editor and for a file-system package, the experiments showed that the better of the two algorithms remained superior to the other one under most of the inputs.

This paper describes an experiment designed to quantitatively evaluate the sensitivity to the input data of the performance improvements resulting from restructuring. Section 2 covers the general design of the experiment and the selection of the experimental factors. In Section 3, the criteria employed to choose the levels of the factors are discussed. The term "level"

is given here the meaning it has in the experimental design literature. The results of the experiment and their interpretation are finally presented in Section 4.

2. The design of the experiment

The main objective of the experiment described in this paper was to determine the sensitivity of the performance improvements induced by restructuring to variations in the input data.

The performance of a restructured program is defined here as the page fault rate F it generates when running on a virtual-memory system. F depends mainly on the following factors:

P: the original program;

R: the restructuring procedure used;

J: the input data chosen for the restructuring procedure;

M: the memory management policy under which the program is run;

I: the input data under which the program is run.

The main goal of the experiment required factor I to be varied. We felt it would be useful to also consider various levels of factor J, in order to determine to what extent the conclusions suggested by the runs corresponding to a particular level of J could be extended to other levels. In other words, our objective was to study the relationship between I and J rather than simply the influence of I on performance.

Ideally, a complete experiment would have required varying all of the five factors listed above. However, such an experiment would have been too expensive to perform. Its cost could have been reduced only by drastically restricting the number of levels considered for each factor including I and J, or by reducing the number of combinations of levels with which the experiment could have been performed. We therefore decided to concentrate

our study on the impact of factors I and J, and of their interactions. Since we were in fact interested in their influence on performance improvement, we considered two levels for R: no restructuring (level R0) and restructuring (level R1). P and M were kept constant throughout the experiment. This obviously means that our conclusions are valid only for the particular constant levels selected for P, R (in the restructuring case) and M. The selections of these levels are discussed in Sections 3.1 and 3.2.

Since we were also concerned with the effects of the interaction between I and J on the improvement of the page fault rate F, the experiment had to be designed as a full factorial one. However, factor J does not play any role in those experimental configurations corresponding to the non-restructuring program. Therefore, our three-factor factorial experiment consisted of $\ell_I(\ell_J+1)$ configurations or distinct combinations of levels, where ℓ_I and ℓ_J represent the numbers of levels of I and J respectively.

Assuming that no experimental errors are made in the measurement of F, the only sources of variation for F are the three factors R, I, J and their interactions. Let R_g be the main effect of factor R at level Rg, I_h the main effect of I at level Ih, and J_k the main effect of J at level Jk. The interaction effect of I and J at levels Ih and Jk will be denoted by IJ_{hk}, and similar notations will be adopted for the other interaction effects. The results of our experiment will be interpreted by the empirical model

$$F_{ghk} = \bar{F} + R_g + I_h + J_k + RI_{gh} + RJ_{gk} + IJ_{hk} + RIJ_{ghk} , \quad (1)$$

where F_{ghk} is the page fault rate of the original (g = 0) or restructured (g = 1) program running under input Ih, and (if g = 1) restructured with input Jk; $\bar{F}$ is the "grand mean", that is the mean page fault rate computed over all the conbinations of levels considered.

The empirical model represented by (1) allows us to break down the total variation of the page fault rate into the contributions of the individual sources of variation. Standard analysis of variance techniques will be applied in Section 4 to the results in order to quantitatively evaluate the relative importance of these sources, in particular that of I and of its interactions with the other factors.

3. The selection of the levels of each factor

3.1 The memory policy and the restructuring algorithm

The restructuring procedure introduced in Section 1 contains several components which, in a more detailed study, could be introduced as additional factors. For instance, the restructuring algorithm, the clustering algorithm and their parameters, the choice and sizes of the blocks, the page size, could all be varied during an experiment, and some of them have in fact been [1-4]. For the study described in this paper, only one component of the restructuring approach was separated from the others and made an independent factor: the input J under which the program is restructured. All the remaining components were fixed for the cases in which the original program had been restructured. Factor R was introduced since some runs of the non-restructured program had to be included in the experiment.

For the case R = R1 (restructured program), the blocks were chosen as described in Section 3.2, and the page size was kept at 1024 words/page. The clustering algorithm used in the experiment is extremely simple: the two nodes in the restructuring graph whose connecting edge has the highest label are clustered together if the sum of their sizes is not greater than the page size; the labels connecting the other nodes to the new node are computed in the obvious way, and the process is repeated until no further clustering is possible. Another algorithm is then employed to order blocks

within clusters and the clusters with respect to each other.

Following the program-tailoring philosophy (see [2] and Section 1 of this paper), the selection of the restructuring algorithm to be used for R = R1 was made together with the choice of the memory policy M, which was also to remain unchanged throughout the experiment. M was chosen to be a "sampled" working set policy. In such a policy, the working set of a program is measured at periodic virtual-time intervals. Those pages which are in the working set at one of these instants are guaranteed to remain in memory until the next sampling time. Those page frames which are occupied by pages not in the working set are returned to the free pool (page writes are ignored for simplicity). New pages are loaded into memory on demand. The memory is initially empty. The window size used to measure the working set was chosen to be equal to the sampling interval, which was 50 ms.

The restructuring algorithm was the one called the A algorithm in [2]. Its objective is to minimize the number of page faults generated by the given program in a sampled working set environment. The sampling interval and the window size for the algorithm were equal to those assumed for M, that is, 50 ms. Thus, the program was restructured trying to tailor its behavior to the one postulated as ideal by the sampled-working-set memory policy [2].

3.2 The program to be restructured

The selection of the program on which the experiment would be performed was made taking a number of criteria into account. First of all, the program had to be relatively large and one which was heavily used in order for our results to have greater practical interest. This is in fact the class of programs for which restructuring is likely to produce the most substantial benefits.

Since an interpreter of the machine language of our computer was not

available, and building one was outside the scope of this project, we decided to restructure only the instruction portion of the program, leaving the data part untouched. On most machines, including ours, an instruction reference string is much easier to obtain than a data or complete (instruction-data) reference string. Proper selection of instruction blocks can facilitate the instrumentation of a program so that the instruction blocks can be traced and time-stamped in a relatively straightforward manner during execution. It is useful to observe that our decision to gather and restructure only instruction-block reference strings does not make the results of the experiment invalid. In a working-set environment, as mentioned in [1], restructuring an instruction-block (or a data-block) string does not cause the results to be distorted with respect to those which would be obtained by restructuring the corresponding complete string, provided that the instructions and data are kept in separate pages. Because of the role played by time in a working-set policy, the two portions of a program (data and instructions) can be restructured independently obtaining the same block orderings and performance improvements as if the whole program were restructured. This is, of course, not the case of other policies such as LRU in a fixed-allocation environment. Thus, restructuring only the instruction blocks is as beneficial, for the instruction portion of a program, as restructuring the same portion according to a full trace of the complete program. The results of our experiment, however, do not represent the full potential of the restructuring procedure used, since substantially greater improvements would presumably be obtained if the data portion of the program were also restructured.

Having decided to concentrate on the instruction blocks, we needed a program whose instruction portion would be sufficiently large and whose blocks would be easy to identify, to rearrange and to instrument. All of the

criteria we have listed indicated that a program like a compiler would be suitable for the single level of factor P. We chose a PASCAL compiler (written in PASCAL) for its understandability and modularity. The compiler has 139 procedures, each one of which was considered as a separate block. The mean size of each block was 129 words with a maximum block size of 669 words and a minimum of 18 words.

Tracing the sequence of execution of the procedures in a program is conceptually straightforward: a trace routine which records the block number and the virtual time is invoked whenever a procedure is entered. The interference caused by this software measurement tool is usually non-negligible. However, the running time of the trace routine was carefully minimized, and the distortion in the virtual times recorded was found to be tolerable over the durations of the runs.

3.3 The inputs to the program

The choice of the levels for factors I and J was the most difficult one. The number of different inputs to be experimented with had to be small enough as to make the experiment economically feasible. On the other hand, it had to provide us with a sufficiently wide spectrum of compilations and compilation performances. We felt that five levels, corresponding to five instrumented executions of the PASCAL compiler, five applications of the restructuring procedure, and thirty runs of the compiler (more precisely, of the five block reference strings recorded during the instrumented runs) in a simulated virtual-memory environment, would be economically acceptable and technically sufficient.

Therefore, five different PASCAL programs had to be selected. How could we measure the magnitudes of their "differences", and what amount of difference would we consider satisfactory? The simplest criterion was that the inputs

have different natures, so that they would exercise the different parts of the compiler as dissimilarily as possible. Based on this criterion, we selected the following five programs:

1. the first 800 lines and the last 400 lines of the PASCAL compiler itself;

2. a program of about 500 statements predominantly consisting of arithmetic operations;

3. the same program as in 2, but with numerous errors inserted in the source code;

4. a global flow analysis program of about 400 source statements;

5. a tree traversal program of about 65 statements.

However, the above criterion was not considered sufficient by itself. The definition of a quantitative measure of the differences between the inputs we had selected was also deemed to be necessary. From the viewpoint of our experiment, two inputs of a program are really different if they cause the program to exhibit substantially different referencing behaviors. The comparison of two referencing patterns is very difficult to perform.

One way of gaining some insight into the question is to compare the reference densities of the various blocks of the program under the two inputs. If the densities are sufficiently different, then the inputs indeed exercise the program differently. Note that this is a sufficient but not necessary condition for two inputs to differ from our viewpoint.

The differences between block reference densities for the PASCAL compiler are reported in Fig. 1. The densities corresponding to input I1 coincide with the horizontal axis in both diagrams. The diagram in Fig. 1(a) shows the densities of inputs I2 and I3, the one in Fig. 1(b) those of I4 and I5. An easier-to-interpret summary of the information displayed in Fig. 1 is presented in Table I, which gives the number of blocks whose

reference densities have a coefficient of variation about the mean density for the same block within a certain range. This table tells us that for 38% of the blocks referenced in at least one run, the standard deviation of the block's reference density is greater than the mean.

Another condition, also sufficient though not necessary, for two inputs to cause different program behaviors is that their respective influences on performance differ by non-negligible amounts. Table II shows the percentage differences between the page fault rates and between the mean working set sizes produced, under policy M and with the non-restructured PASCAL compiler, by the five inputs listed above.

The results displayed in Fig. 1 and Tables I and II convinced us that the five levels selected for factors I and J were sufficiently different from each other for the purposes of our experiment.

4. The results of the experiment

The mean page fault rates measured during the 30 runs of which the experiment consisted are reported in Table III. As mentioned in Section 3, the runs were performed by trace-driven simulation. The simulated environment represented a virtual-memory system in which the memory hierarchy is managed by a sampled working set policy with a window size of 50 ms. The total CPU times required to compile the five input programs were, respectively, 163.51 s, 52.88 s, 50.49 s, 34.07 s, and 5.05 s. The mean page fault rates displayed in Table III were computed as the ratios between the number of page faults generated by a string and the total simulated time corresponding to the same string.

The effect of R on the page fault rate evidently dominates the effects of the other factors in Table III. The analysis of variance allows us to quantify this observation: as shown in the pie-chart in Fig. 2, more

than 86% of the total variation about the grand mean is to be attributed to factor R. The other two factors are individually responsible for minor portions of the total variation: I only explains the 1.86%, and J the 0.27%. Their interaction has a slightly larger impact (5.41%), and so does the interaction RIJ of all factors.

The results of the analysis of variance are reported in Table IV. If we apply the F-test to them, we find, not surprisingly, a confirmation of the above conclusions. The hypothesis that R has no effect is rejected at a level of significance well below 1%, whereas the one that all the other sources of variation have no effect cannot be rejected, even if the selected significance level is as large as 25%.

5. Conclusion

The experiment which has been described in this paper has quantitatively shown that the performance improvements due to restructuring are not appreciably influenced by the inputs to the restructured program. In other words, the improvements are "robust" with respect to input variations.

This conclusion is, to be precise, valid only for our PASCAL compiler, the range of inputs covered by the five programs we chose, the memory policy and the restructuring algorithm used in the experiment. How extendible the conclusion is to other levels of the factors which were kept constant is anybody's guess. The experience of all those who have published the results of empirical studies of restructuring algorithms, including the authors, suggests that the only factor likely to have a significant influence on the improvements due to restructuring is P, the program to be restructured. If this program is already well-structured in the referencing-behavior sense, only minor improvements can be obtained by restructuring it. In this case, the variations due to the input will tend to be more significant than when

restructuring produces a substantial improvement. Our feeling is that, when a program's performance is so enhanced by restructuring as to make the application of a restructuring procedure really worthwhile, the inputs do not usually have a major influence. Unfortunately, this feeling can only be substantiated by further experimentation.

Acknowledgments

The authors are grateful to David Ching for his valuable help in designing and performing the experiment.

References

[1] D. Ferrari, Improving locality by critical working sets, Comm. ACM 17, 11 (November 1974), 614-620.

[2] D. Ferrari, Tailoring programs to models of program behavior, IBM J. Res. Dev. 19, 3 (May 1975), 244-251.

[3] D.J. Hatfield and J. Gerald, Program restructuring for virtual memory, IBM Sys. J. 10, 3 (1971), 168-192.

[4] T. Masuda, H. Shiota, K. Noguchi and T. Ohki, Optimization of program organization by cluster analysis, Information Processing 74 (Proc. IFIP Congress 74), North-Holland Publishing Co., Amsterdam, 261-265.

[5] D. Ferrari, Improving program locality by strategy-oriented restructuring, Information Processing 74 (Proc. IFIP Congress 74), North-Holland Publishing Co., Amsterdam, 266-270.

Table I

Distribution of the coefficient of variation
of the block reference densities for the PASCAL compiler

Range of the coefficient of variation	Number of blocks
0-20%	8
20-40%	27
40-60%	8
60-80%	20
80-100%	13
100-120%	13
120-140%	19
140-160%	5
160-180%	2
180-200%	3
200-220%	6
Non-referenced blocks	15
Total	139

Table II

Percentage differences between the page fault rates (mean working set sizes)
produced by the five inputs selected

Input	Page fault rate [faults/s]	Working set size [pages]	I2	I3	I4	I5
I1	35.7348	5.44	1.8 (51)	0.14 (43)	5 (54)	4.6 (61)
I2	36.3808	8.23	--	1.6 (1.4)	3.1 (1.9)	6.3 (6.8)
I3	35.7857	8.11	--	--	4.9 (3.4)	4,8 (8.3)
I4	37.5403	8.39	--	--	--	9.2 (4.7)
I5	34.0594	8.79	--	--	--	--

Table III

The results of the experiment
(page fault rates in faults/s)

		Factor I				
Factor R	Factor J	I1	I2	I3	I4	I5
R0	--	35.7348	36.3808	35.7857	37.5403	34.0594
	J1	23.5949	23.7685	24.2994	24.0387	19.6039
	J2	23.0689	14.6733	17.5067	25.7998	28.7128
R1	J3	23.1056	15.5431	15.5461	27.8544	23.7623
	J4	22.0659	23.9387	24.9332	17.9336	19.0099
	J5	21.6867	21.6696	23.5072	22.6005	14.2574

Table IV

Analysis of variance of the results in Table III

Source of variation	Sum of squares	Degrees of freedom	Mean square
R	2520.8266	1	2520.8266
I	54.4766	4	13.6191
J	7.8350	4	1.9587
RI	20.5616	4	5.1404
RJ	7.8350	4	1.9587
IJ	158.3627	16	9.8976
RIJ	158.3632	16	9.8977
Total	2928.2607	49	

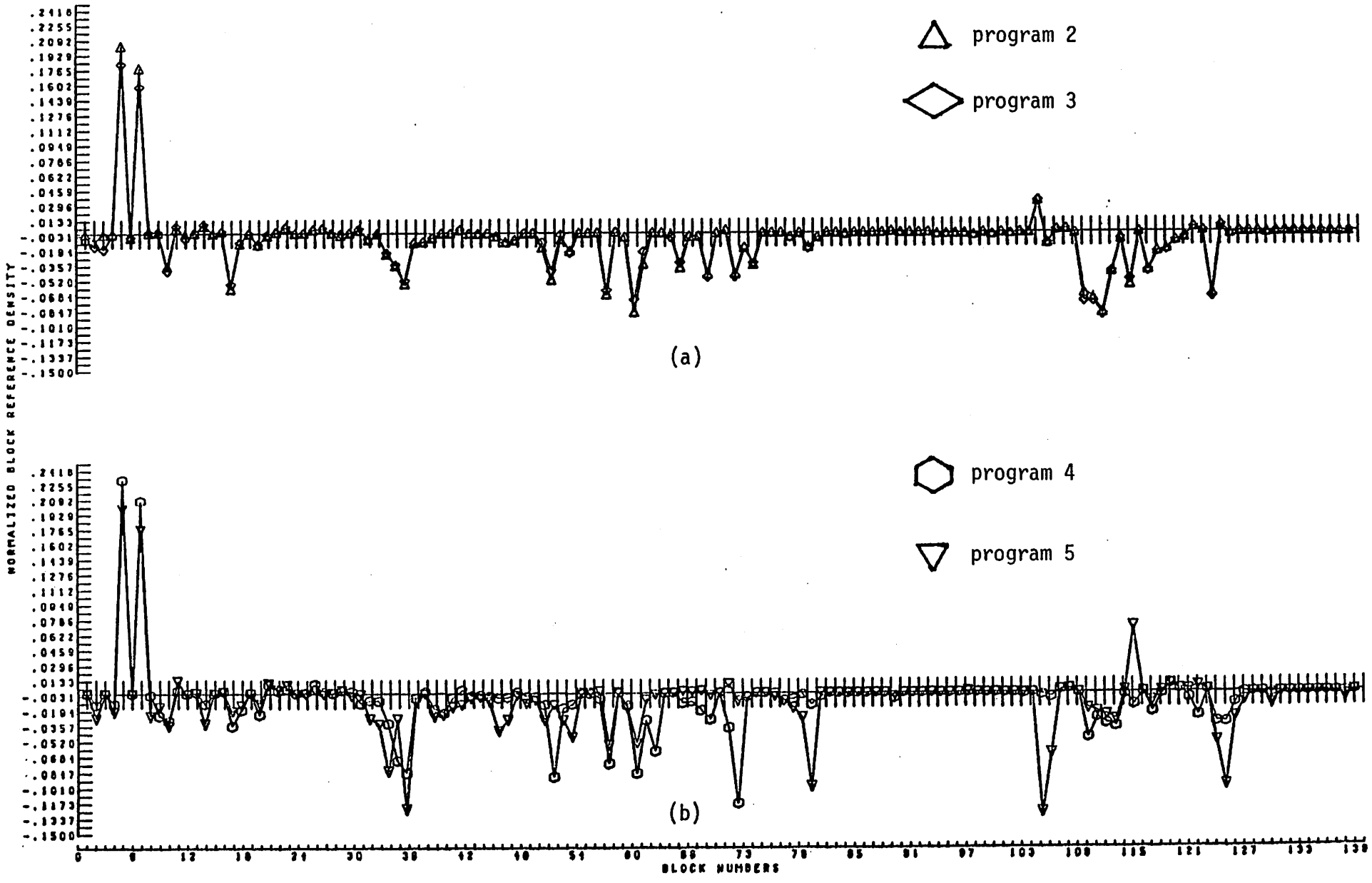

Fig. 1. Differences between the block reference densities of the PASCAL compiler compiling programs (a) 2, 3, (b) 4, 5 and of program 1.

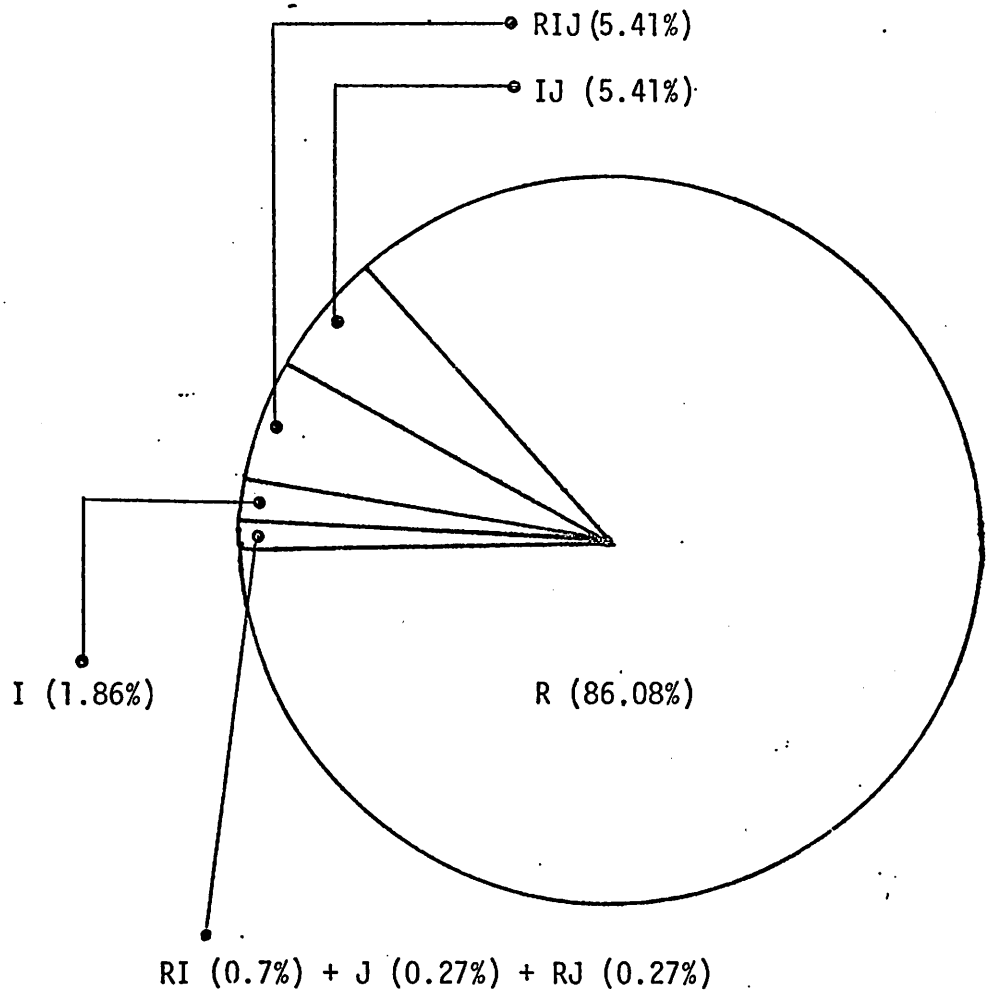

Fig. 2. The influence of the various sources on the total variation of the page fault rate.