STRATEGY-INDEPENDENT PROGRAM RESTRUCTURING BASED ON BOUNDED LOCALITY INTERVALS

by

Samuel T. Chanson & Bernard Law

TECHNICAL REPORT 81-9

Strategy-independent Program Restructuring based on Bounded Locality Intervals

by

Samuel T. Chanson & Bernard M. Law

Dept. Of Computer Science,
University of British Columbia, August 1981.

ABSTRACT

A new program restructuring algorithm based on the phase/transition model of program behaviour is presented. The scheme places much more emphasis on those blocks in the transition phases in the construction of the connectivity matrix than the existing algorithms. This arises from the observation that the page fault rate during the transition phases is several orders of magnitude higher than that during the major phases. The strategy is found, for our reference strings, to outperform the critical working set strategy (considered to be the current best), by non-negligible amounts. Furthermore, the overhead involved is lower than that of CWS and not much higher than that of the Nearness method which is the simplest scheme known. Being strategy-independent, it also seems to respond better than CWS when the memory management strategy used is not the working set policy.

1. Introduction

In a virtual memory system, the performance of a program in execution is strongly influenced by the way its instruction codes and data are distributed among the levels of the memory hierarchy, and how this information is accessed. This is one of the reasons for the interest in program behaviour --- the study of the mechanism underlying the observed memory reference pattern of a program.

Numerous measurement experiments have been performed to study memory reference behaviour. A commonly observed property is that a program in execution favours a subset of its segments (or blocks) during extended periods of time [3,5,6,12,17,19,21]. This property of reference clustering has come to be known as program locality or locality of reference. It is also well-known that by improving the degree of locality of the program through reorganization of its relocatable blocks in the virtual address space, substantial improvements in the paging behaviour of the program can be obtained (see for example [1,4,8,15,16,20]). This approach is known as program restructuring. The objective is to rearrange the blocks of a program such that those that are needed within a relatively short time of one another are found either in the same virtual page or in pages that would otherwise tend to be in physical memory at the same time. This has the obvious effect of reducing the page fault rate. It has also been shown that

program restructuring is cost-effective for certain types of programs, particularly system programs such as compilers and editors (see for example [15]).

An initial requirement for programs to benefit from restructuring is that they must consist of many relocatable blocks the size of which should be small compared to the page size. Hatfield and Gerald [16] for example, suggest that the mean block size should be between a tenth to a third of the page size.

The procedure of dynamic restructuring of programs normally consists of the following steps:

Step 1. Based on the string of block references during program execution, the desirability of pairing blocks together is computed. This information is often stored in a matrix. Thus the element (i,j) of the "desirability" (or connectivity) matrix represents the desirability of placing blocks i and j in the same page. The various restructuring methods reported in the literature differ mainly in the way the connectivity matrix is computed.

Step 2. The blocks with the strongest connectivities are grouped into clusters with the restriction that a cluster must fit into a single page. Clustering techniques are well-known (see for example [1,19]) and will not be discussed here.

Step 3. The clusters are mapped into pages. Since it is

sometimes not desirable to leave unused memory space between clusters, various sequencing methods have been proposed to order the clusters (see for example [19,24]).

In this paper, we shall focus our attention on the restructuring algorithms (i.e., those used to compute the connectivity matrix in Step 1).

All of the existing restructuring algorithms are based on the raw (unaltered) block reference strings. Thus, generally blocks that are frequently referenced will have strong connectivities. This may seem to produce the desired result but let us examine the program reference pattern in some detail.

2. Program behaviour

Various studies have shown that program behaviour can be reasonably represented by the phase/transition model [11,17,21,22,23]. From the viewpoint of program locality, a program's execution time can be regarded as a sequence of locality phases (or simply phases) separated by transitions. Informally, a phase is an interval during which the small set of blocks being referenced is constant; and a transition denotes the interval of migrating from one phase to another. By subdividing the program into blocks, the locality set of a phase (phase set) is the set of blocks active in that phase. (A given

block is considered active in a phase whenever processing of that phase requires the presence of the block in main memory.) Similary, the transition set is the set of blocks active during the transition.

Kahn's empirical studies [17] show that the phases cover at least 98% of virtual time but the disruptive transition intervals between the phases dominate the page fault behaviour. The rate of page faults during transitions was found to be 100 to 1000 times higher than that during phases. Moreover, the work of Denning and Kahn [11] indicates that for executable memory size greater than the mean phase size (which is normally the case), performance is much more dependent on the characteristics of transitions than either the program behaviour within phases or the memory management algorithm in use.

Thus by using the raw block reference string to compute connectivities, the blocks in the transition sets are unduly ignored. This is because though they cause the majority of the page faults, they are referenced comparatively infrequently. Therefore their connectivities must necessarily be very low compared to those of the blocks in the phase sets. We believe that an effective restructuring algorithm should put more emphasis on the transition blocks.

An obvious method is to identify the transition sets and to use more weights for these blocks in the computation of the

connectivity matrix. However, a basic difficulty with the phase/transition model is the one of formulating a procedure which could identify all the distinctive phases and their corresponding locality sets given a reference string [11]. The notion of the Bounded Locality Interval (BLI) introduced by Madison and Batson [18] overcomes this problem.

The initial idea of the BLI was triggered by the observation that the least-recently-used (LRU) stack contains, at any time t, the blocks arranged in the order of the times at which they were last referenced, with the most-recently-used block at the top of the stack. If the top i elements in the LRU stack are {Pi} then we can also record fi, the time of formation of this set and ei, the time at which a reference was last made to a stack position greater than i. (Thus ei is the termination time of {Pi}.) At any instant t, an activity set is defined as any {Pi} in which every element of that set has been referenced more than once since the set has been formed at the top of the LRU stack. The lifetime, li, of such an activity set is defined to be the difference between fi and ei where ei > fi. A BLI is the pair (Ai,li), such that Ai is the activity set and li is its lifetime (see [18] for details).

The notion could be generalized by defining an activity set as one whose elements have been referenced at least k times since the set was formed. In particular, the definition given above is for the case when k=1. Moveover, k is the only

parameter of the model and it is independent of the memory management policy. Another important characteristic of the BLI is the implicit hierarchical nature of localities embedded in the definition. More on this in the next section.

The concept of the BLI is a reasonable solution to the problem of identifying distinctive major phases in a given reference string, except that we still have to determine whether a phase is really a "major" phase. Intuitively one would expect that a major phase should have a reasonably long lifetime. Batson [2] correctly points out that the criterion of "reasonably long" can only be formulated in the context of the particular virtual memory system upon which the program will be executed. He suggests that the mean time required to transfer a block from secondary storage to main memory can be used to determine a sequence of major phases and transitions. If p is the block transfer time, each BLI is regarded as a major phase if its lifetime is greater than p.

3. The restructuring algorithm

We shall call the new algorithm the BLI restructuring algorithm because of the importance of the BLI notion in its formulation. The given reference string is first partitioned into a sequence of major and transition phases using the definition of BLI with a chosen value for the parameter k and

the block transfer time p. To keep the cost down, one may want to use only the top level BLI's, which partition the reference string into the longest possible subintervals of distinctive referencing behaviour. However, our empirical results show that if the phases are too long and consist of more blocks than can be accommodated in a page, the results are not as good as those using lower level (and therefore smaller) BLI's. This is because the lower level BLI's provide more detailed information about the reference pattern. We choose to use the lowest level (i.e., the shortest) BLI's whose length exceeds p. It turns out that because of the way the connectivity matrix is constructed, this does not increase the overall computational overheads of the scheme.

For each of the phases, we obtain an unordered set consisting of all the distinct blocks in the phase set. The connectivity matrix M=[mij] which is an nxn matrix (n is the total number of blocks in the program) with indices labelled by block numbers is constructed as follow. M is initialized to zero. For each set of blocks and for all combination of block pairings of i and j in the set, mij is incremented by one where i is different from j.

Notice that unlike the other existing algorithms, a block in a transition set is given the same weight in the construction of the connectivity matrix despite its much lower frequency of reference during execution. This is a direct consequence of

mapping the reference string of each phase (major or transition) into a set of distinct blocks. Since every block in a set appears exactly once, the frequency of reference is taken to be the same for all blocks.

It should be apparent that a whole family of program restructuring algorithms based on BLI is possible. First, k can assume different values (the value of p is fixed by the characteristics of the paging device). Then when the major and transition phases are identified, one can use i) only the major phase sets, ii) only the transition phase sets or iii) both the major and transition phase sets in the computation of the connectivity matrix. Since i) defeats the purpose of the BLI restructuring policy which emphasizes the contribution of the transition phases to page fault behaviour it will not be considered further. Noting that a transition set generally also contains the blocks in the next major phase set, ii) then is similar to iii) except that it places even more weights on the blocks in the transition sets. Our initial experiments show that ii) and iii) produce very similar results. Since ii) is less expensive , it is used throughout the subsequent experiments that are reported in this paper.

Finally, one could order the blocks in each set according to their first reference times and consider only adjacent pairs (or combinations of j consecutive references, j=2,3,...) in constructing the connectivity matrix. Our empirical results

show that due to the small set size (about 5 for major and 10 for transition sets), it is more effective and not much costlier to consider all combinations of pairs in each set. The latter method provides a more global view of the reference pattern.

3. Description of the experiment

The experiment is based on block reference strings gathered from the execution of a Pascal compiler on an Amdahl 470 V/6 II computer. The Pascal compiler was written in Pascal and considered to be well structured. It consisted of 336 relocatable blocks (procedures and functions) and about 90% of these had sizes less than 800 bytes. The average block size was about 600 bytes. Thus each 4K-byte (4096) page could hold about 7 blocks on the average. The compiler consisted of about 54 pages of codes. The block reference strings collected are 'reduced' in the sense that successive references to the same block produce only a single reference. (Thus for example, aaaaabbbcccc yields simply abc.)

Six programs were used as input to the Pascal compiler in the experiment. They ranged from production programs to artificial ones consisting of various number of syntax errors. The performance of three restructuring algorithms - Hatfield and Gerald's Nearness method (NEAR) [16], Ferrari's Critical Working Set (CWS) [13] (considered to be the current best) and the BLI

algorithm were compared. The clustering algorithm used was the one of hierarchical classification [19]. The mapping of clusters into pages followed the same procedure as used in CWS [13]. Since CWS is a strategy-oriented restructuring algorithm and assumes the working set memory management policy [9] to be used, the window size is also a factor in our experiments. To reduce the number of factors, various window sizes, ranging from 10 to 100 block references were tested using one of the reference strings. The window size of 50 was found to give the best performance for all three restructuring algorithms in terms of the percentage reduction in page fault rate relative to that of the original ordering (ORIG). This value was subsequently used in all the experiments. The factors and their values (known as levels) are listed in Table 1.

The performance indices are chosen to be the page fault rate and the mean working set size. Together they cover, to certain extent, the space and time components of a computational activity.

Since the absolute improvement is influenced by the values of the other factors used in the experiments as well as by the quality of the layout in the original program, the results must be interpreted as such. The magnitude of the relative performance improvements of both NEAR and CWS are consistent with those reported in the literature.

TABLE 1. FACTORS AND LEVELS FOR THE EXPERIMENT

PROGRAM: A PASCAL COMPILER
MEMORY MANAGEMENT POLICY: WORKING SET

FACTORS	NAME	LEVELS DESCRIPTION
Input data (program to be compiled)	P1	program P, 25 statements, 50 errors
	P2	program P, 25 statements, 25 errors
	P3	program P, 25 statements, 5 errors
	P4	program P, 25 statements, no errors
	P5	program Quicksort, 60 stts, no errors
	P6	program BLI,355 statements, no errors
Restructuring Algorithms	ORIG	Original ordering
	NEAR	Hatfield's Nearness Method
	CWS	Ferrari's Critical-Working-Set
	BLI	k = 1, p = 15
Clustering Algorithms	NUCL	Nucleus-constructing
	HIER	Hierarchical classification (HIER found to be superior. Subsequent value fixed at HIER)
Window size (references)	10 to 100 in increments of 10	50 was found to be optimal for all restructuring algorithms (see Section 3). Subsequent value fixed at 50.
Page size		Fixed at 4096 bytes

4. Results

Improvements in the number of page faults and the average working set size by program restructuring are computed in terms of percentage reduction according to the formula

$$\% \text{ Reduction} = (Po - Pr)/Po * 100$$

where Po and Pr are the original and restructured performance indices respectively.

Table 2. Comparsion of the 3 restructuring algorithms to ORIG on percentage reduction in the number of page faults.
Memory policy: working set

ref. string	# block ref.*	# page faults (ORIG)	restructuring algorithms NEAR	CWS	BLI
P1	10950	606	23.4	32.3	35.8
P2	9951	693	25.3	33.2	39.5
P3	21305	1764	32.4	36.7	41.6
P4	7216	533	26.1	31.5	38.1
P5	34802	3066	30.8	33.4	42.5
P6	62194	5507	25.9	35.2	39.4
mean over all strings			27.3	33.7	39.5

* successive references to the same block produce only 1 block reference (see Section 3).

Table 3. Comparsion of the 3 restructuring algorithms to ORIG on percentage reduction in the mean working set size. Memory policy: working set

ref. string	mean working set size in pages (ORIG)	restructuring algorithms		
		NEAR	CWS	BLI
P1	5.9	27.5	31.6	28.2
P2	6.8	27.2	27.8	27.9
P3	7.5	30.3	26.6	29.5
P4	6.8	24.7	30.4	26.3
P5	7.6	28.2	25.2	27.2
P6	7.9	24.8	26.9	28.9
mean over all strings		27.1	28.0	28.0

Table 4. Comparsion of the 3 restructuring algorithms to ORIG on percentage reduction in the number of page faults. Memory policy: first-in-first-out

ref. string	# page faults (ORIG)	restructuring algorithms NEAR	CWS	BLI
P1	396	40.2	52.3	61.6
P2	457	35.9	45.5	55.1
P3	1121	56.6	64.9	80.2
P4	384	33.1	53.9	58.9
P5	2088	47.4	50.6	73.6
P6	3572	43.5	55.0	68.7
mean over all strings		42.8	53.7	66.4

Table 5. percentage reduction in the number of page faults over ORIG using the layout for P1 over all other strings. Memory policy : working set

restructuring algorithm	reference string P2	P3	P4	P5	P6	mean
NEAR	20	15	19	18	15	17
CWS	26	19	24	22	23	23
BLI	30	23	29	26	23	26

Table 2 shows that BLI is superior to CWS for all six reference strings, averaging 17.2% over CWS in the reduction of the number of page faults. CWS, on the other hand, is 23.4% better than NEAR which in turn yields 27.3% less page faults than the case of no restructuring. As is evident from Table 3, there does not seem to be any substantial difference in the mean working set size among the three algorithms. Each is about 28% better than the case of no restructuring.

To test the performance of BLI under memory management other than the working set policy, the experiment corresponding to Table 2 was repeated, this time using a fixed partition (15

page frames) first-in-first-out page replacement policy. The results are presented in Table 4. We note that while the relative improvement of CWS over NEAR remains at about 25%, BLI is now 23.6% better than CWS instead of 17.2%. This may indicate the strength and portability of the BLI scheme, which is strategy-independent, over the tailored restructuring algorithms (see also [15]).

Finally, a set of experiment was performed to test the data dependence of BLI. Table 5 shows the percentage reduction in the number of page faults relative to ORIG for reference strings P2 through P6 using the layout corresponding to the reference string P1. The memory policy is the one of working set. There is a general degradation of relative improvement for all three schemes, but the BLI method is still superior to CWS in all cases, averaging 26% improvement over ORIG and 13% over CWS. CWS is now 35% better than NEAR which would suggest that NEAR is most data dependent and CWS is the least data dependent of the three algorithms.

The cost of program restructuring basically consists of the costs in gathering the reference string, constructing the connectivity matrix and clustering. The cost of gathering the reference string is independent of the restructuring algorithm. As well, the cost of clustering in our experiments do not differ noticably among the three algorithms. The cost of constructing the connectivity matrix in terms of the CPU time used is least

for NEAR which is the simplest of the three. That for CWS was found to be about three times as much. For BLI, the cost was less than half that for CWS and just 30% above that for NEAR. The main reason is the small size of the reduced major phase and transition sets (about 5 and 10 blocks respectively).

A simple algorithm has been derived which would locate the approximate major and transition phases with less overheads than the scheme outlined by Madison and Batson [18]. For the purpose of program restructuring, it is perhaps even more suitable since it is non-hierarchical and, for our reference strings, generates reasonably small major and transition phase sets. The experiments reported here have been repeated using this method of identifying the localities and the results are always within $\pm$5% of the BLI results. The mean of the results are practically the same as that for BLI. The algorithm can be described as follows. Start off by gathering the blocks into a transition phase set. A major phase starts as soon as a block in the transition phase set is referenced again and it ends when a block not belonging to the transition phase is referenced. Of course, a phase is not considered a major phase if its length does not exceed p. (Recall that in gathering our block reference strings, successive calls to the same block are replaced by a single reference.)

The CPU time in constructing the connectivity matrix using this algorithm is only about 13% higher than the one for NEAR.

5. Conclusion

A new program restructuring algorithm based on the phase/transition model of program behaviour has been described. The scheme places much more emphasis on those blocks in the transition phases in the construction of the connectivity matrix than the existing algorithms. This arises from the observation that the page fault rate during the transition phases is several orders of magnitude higher than that during the major phases. The strategy is found, for our reference strings, to outperform the critical working set strategy (considered to be the current best), by non-negligible amounts. Furthermore, the overhead involved is lower than that of CWS and not much higher than that of the Nearness method which is the simplest scheme known. Being strategy-independent, it also seems to respond better than CWS when the memory management strategy used is not the working set policy.

ACKNOWLEDGEMENT

This work was supported in part by the National Science and Engineering Research Council of Canada under grant A3554.

REFERENCES

[1] M. S. Achard, J.Y.Babonneau, M.Carpentier, G.Morisset, and M.B.Mounajjed, "The Clustering Algorithms in the Opale Restructuring System," Performance of Computer Installation, D. Ferrari(ed.) CILEA, North-Holland Publishing Co., pp.137-153, June 1978.

[2] A. P. Batson, "Program behavior at the symbolic level," Computer, vol.9,no.11,pp.21-28, Nov. 1976.

[3] L. A. Belady, "A study of replacement algorithms for virtual storage computers," IBM Syst. J., vol.5, no.2, pp.78-101, 1966.

[4] L. A. Belady, and C. J. Kuehner, "Dynamic space sharing in computer systems," Commun. ACM, vol.12, pp282-288, May 1969.

[5] B. Brawn and F. G. Gustavson, "Program behavior in a paging environment," in 1968 AFIPS Conf. Proc., Fall Joint Comput. Conf., vol.33, Washington, DC, 1968, pp.1019-1032.

[6] W. W. Chu and H. Opderbeck, "The page fault frequency replacement algorithm," Proc. FJCC, pp.597-609, 1972.

[7] W. W. Chu and H. Opderbeck, "Program behavior and the page fault frequency algorithm," IEEE Computer 9 11, Nov. 1976, pp.29-38.

[8] L. W. Comeau, "A study of the effect of user program optimization in a paging system," in Proc. ACM Symp. Operating Systems Principles, Oct. 1967.

[9] P. J. Denning, "The working set model for program behavior," Commum. ACM, vol.11, pp.323-333, May 1968.

[10] P. J. Denning and G. S. Graham, "Multiprogramming memory management," IEEE Proc., vol.63, pp.924-939, June 1975.

[11] P. J. Denning and K. C. Kahn, "A study of program locality and lifetime functions," in Proc. 5th ACM Symp. Operating Systems Principles, pp.207-216, Nov.1975.

[12] P. J. Denning, "Working Sets Past and Present" IEEE Trans. on Software Engineering vol.SE-6,no.1, pp.64-84 Jan. 1980.

[13] D. Ferrari, "Improving Locality by Critical Working sets," CACM, vol.17, pp.614-620, Nov. 1974.

[14] D. Ferrari, "Improving Program Locality by Strategy-Oriented Restructuring," Information Processing 74, Proc. IFIP Congress 74, North-Holland, Amsterdam, pp266-270, 1974.

[15] D. Ferrari, "The improvement of program behavior," IEEE Computer 9 11, pp.39-47, Nov.1976.

[16] D. J. Hatfield and J. Gerald, "Program restructuring for virtual memory," IBM Sys. J. vol.10, no.3, pp.168-192, 1971.

[17] K. C. Kahn, "Program behavior and load dependent system performance," Ph.D. Dissertation, Dept. of Computer Sci., Purdue Univ., W. Lafayette, IN, Aug. 1976.

[18] A. W. Madison and A. P. Batson, "Characteristics of program localities," CACM, vol.19, pp.285-294, May 1976.

[19] T. Masuda, H. Shiota, K. Noguchi, and T. Ohki, "Optimaztion of program locality by cluster analysis," in Proc. IFIP Congress, pp.261-265,1974.

[20] T. Masuda, "Methods For the Measurement of Memory Utilization and the Improvement of Program Locality", IEEE Trans. on Software Engineering, vol.SE-5,no.6, pp.618-631, Nov.1979.

[21] J. R. Spirn and P. J. Denning, "Experiments with program locality," in AFIPS Conf. Proc., FJCC, vol.41,

Montvale, NJ: AFIPS Press, pp.611-621,1972.

[22] J. R. Spirn, Program Behavior: Models and Measurement. New York: Elsevier/Noth-Holland, 1977.

[23] G.S. Graham, "A Study of Program and Memory Policy Behaviour," (Ph.D. Thesis), Purdue University, Computer Science Dept, 1976.

[24] R. N. Horspol and J. M. Laks, "An improved block sequencing method for program restructuring," Dept. of Computer Science, McGill University, April 1981.