Code Reorganization for Instruction Caches[*]

A. Dain Samples[†]

Paul N. Hilfinger

Computer Science Division–EECS

University of California at Berkeley

Berkeley, CA 94720

October 18, 1988

Abstract

Program performance can be improved on machines with virtual memory by reorganizing the program's address space. We address the question whether reorganization could benefit programs on machines with instruction caches. We have performed experiments to determine the efficacy of restructuring using simple reordering algorithms and profile data, concluding that performance improvement can be obtained relatively cheaply. Experiments show improvements in miss rates on the order of 30% to 50%, and sometimes as high as 50% to 80%, by performing a simple algorithm that relocates only 3% to 8% of the basic blocks of a program.

[*]This research was sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292.

[†]Supported in part by an AT&T Bell Laboratories Scholarship.

1 Introduction

There have been many investigations looking to improve computer performance by reorganizing programs' address spaces on machines with virtual memory [1,2, 3,4,5,9,10,11,13,20,22,23]. We address the question whether reorganization can be beneficial for machines with caches. After all, a cache is just another layer in the memory hierarchy of a computer system. If an inexpensive way can be found to reorganize the address space of a program such that a small cache can have the performance of a larger cache without reorganization, smaller inexpensive caches would be a more competitive choice.

The research reported here was done in a broader context of what we are calling profile-driven optimization. The idea is not new. In his paper on the empirical properties of FORTRAN programs, Knuth [18] proposed an "ideal system" that would keep profiles associated with source programs and use them to direct debugging and optimization.

With the memory capacity of today's desk-top computers matching that of the supercomputers of the 70's, compilers will begin to take advantage of more and varied data to aid in the construction of quality programs. This data will come from many sources, and one is bound to be the behavior of programs themselves. Assuming that a compiler has access to profile data about the program being compiled, we are looking for simple algorithms that will improve compiled programs' performance with respect to an instruction cache. In this study, the only profile-driven optimization we allow the compiler is reorganization of the instruction space.

Instructions that are executed close together in time are *temporally* local. Instructions that are close together in the address space are *physically* local [7]. A cache turns temporal locality into physical locality by effectively and transparently changing the hardware address of data or code. We will show how to utilize information about the runtime behavior of a program to enhance the performance of an instruction cache.

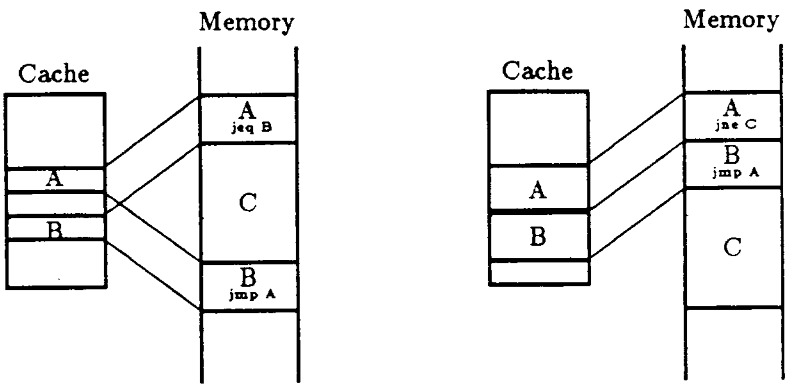


Figure 1: The code at block A and the code at block B are the active portions of a loop. Due to intervening code C that is infrequently executed, A and B are mapped to the same locations in the direct mapped cache, as shown on the left. The loop can be made more efficient by moving A and B with respect to one another so that they do not conflict in the cache, as shown on the right.

There are only two ways to improve the performance of a program in a cache: (1) decrease the probability that often executed sections of the program compete for cache resources (Figure 1); and (2) increase the amount of useful information in the cache (Figure 2).

For a fully associative cache there may be ways of reorganizing a program to improve its performance with respect to (2); little can be done as far as (1) is concerned. With direct mapped caches, however, both (1) and (2) suggest easy ways to gain performance improvement. In a direct mapped cache, contention is a function of the addresses of the competing program segments, which is easily controlled by a loader and/or compiler.

While there have been published results for organizing data in memory to improve cache performance [24] [8], there has been little published regarding rearranging the instruction space. An exception is a recent report on Scott

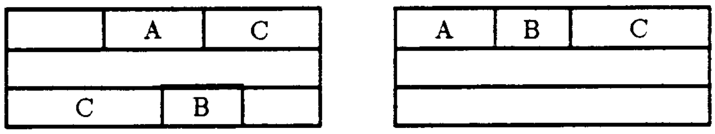


Figure 2: Consider the blocks A, B, and C from the previous figure to be of such a size that A and B could fit in a cache line. The cache lines on the left represent one way they might fit into a direct mapped cache, with the side-effect of loading infrequently executed code from block C into both lines. In the cache on the right the initial cache miss that loads the code from A also loads B, saving at least one cache miss in the execution of the loop; also, infrequently executed code from C takes up much less space.

McFarling's work at Stanford [21]. His work differs from ours by concentrating on positioning basic blocks based on their frequency counts and with knowledge of the target cache. Since direct-mapped caches are cheaper and easier to build [16], we have developed an algorithm for direct mapped caches that uses arc counts and is independent of the target cache.

2 Greedy Sewing

The basic algorithm is quite simple, and is given in Figure 3 below. We require a few definitions. A program control-flow digraph is a tuple $< \mathcal{B}, \mathcal{A} >$ where $\mathcal{B}$ is the set of basic blocks and $\mathcal{A}$ is the set of directed arcs connecting them. We will use upper case letters to denote nodes of the basic block digraph. Basic block A has an arc to basic block B if the flow of control out of A can go to B either by jumping to B, calling B, or simply falling through to it. Arcs will be represented by lower case letters or by the basic blocks they connect; if

```
$s \leftarrow 0$
**repeat**
    *Select* $a \in \mathcal{A}$ *such that* $a_c$ *is maximum.*
    $\mathcal{A} \leftarrow \mathcal{A} - \{a\}$
    $s \leftarrow s + a_c$
    **If** $canSew(source(a),\ sink(a))$ **then**
        $Stitch(source(a),\ sink(a))$
    **If** $isSmallEquiCondl(source(a))$ **then**
        $ThreadCondl(source(a))$
    **else if** $isCall(source(a))$ **then**
        $ThreadCall(source(a))$
    **until** $s > \mathcal{S}p$
```

Figure 3: (*Greedy Sewing Algorithm*). This algorithm orders the basic blocks based on arc traversal frequencies. Its one parameter is a number p in the range $(0,1]$. $\mathcal{A}$ is the set of arcs.

$a = A \rightarrow B$ then we define source(a) = A and sink(a) = B. Associated with each arc a (basic block B) in the graph is a positive integer a_c (B_c) representing the number of times this arc (basic block) was traversed during the profiled execution of the program. The count on an arc is also represented as $A \xrightarrow{c} B$. Let $\mathcal{S} = \sum_{a \in \mathcal{A}} a_c = \sum_{B \in \mathcal{B}} B_c$ be the sum of these counts for all arcs in $\mathcal{A}$ and basic blocks in $\mathcal{B}$.

Stitch(A, B) is a function that juxtaposes two basic blocks contiguously in memory; in this case we say that the *bottom* of A has been sewn to the *top* of B, and that they are on the same *thread*. A thread is a sequence of basic blocks that have been *Stitch*ed. The final instructions in A are adjusted to maintain the semantics of the original program, as shown in the example in Figure 1. If *Stitch* is called and either the bottom of A or the top of B has already been *Stitch*ed, it returns without effect. The function *canSew(A, B)*, tests for this condition for basic blocks A and B.

In Figure 3 there are four functions, *isSmallEquiCondl*, *isCall*, *ThreadCondl* and *ThreadCall*, that handle two situations that arise frequently enough to

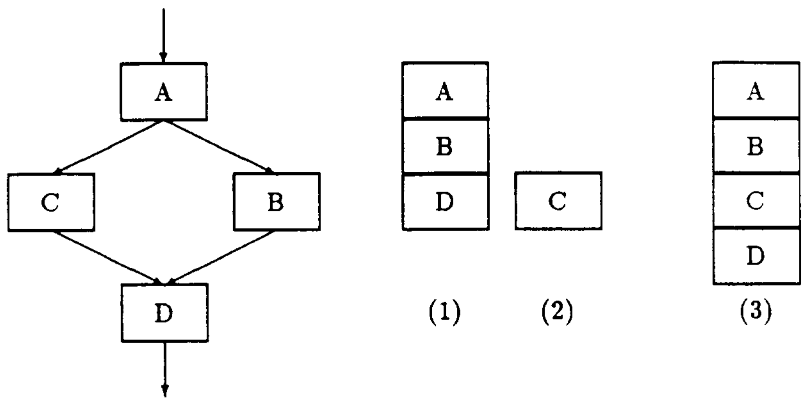


Figure 4: Two threads from if-then-else

warrant special handling. Consider the flow graph in Figure 4. If the path $A \rightarrow B \rightarrow D$ is the more frequently executed path, then the thread ABD will be formed (1). Since C cannot be sewn to either A or D now, it creates a singleton thread (2). If it is very infrequently executed, then it makes little difference where C is placed relative to the thread ABD. However, if the path $A \rightarrow C \rightarrow D$ is only slightly less frequently executed than the path $A \rightarrow B \rightarrow D$, and if the sum of the sizes of A, B, C, and D are small enough that they would all fit in a cache, then the single thread $ABCD$ in Figure 4(3) is preferable over the two threads (1) and (2). Since we want our algorithms to be general and not depend on any particular cache configuration or size, we cannot know whether any set of basic blocks will or will not fit in a cache. So *isSmallEquiCondl* checks for basic blocks matching exactly this configuration—i.e. the basic block ends with a conditional instruction, the two arms of the conditional have at most one basic block in them and are very nearly equi-probable—and creates the longer thread where possible.

A second common situation is pictured in Figure 5 where a procedure P is called from a basic block A. When the bottom of A is sewn to the top of P_0, we

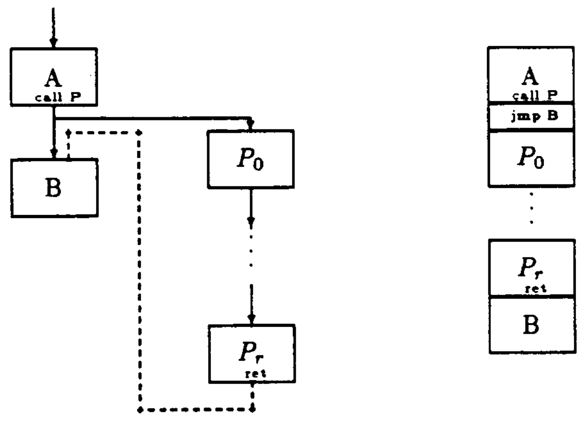


Figure 5: Pseudo-inlining

say that procedure P has been *pseudo-inlined.* A basic block containing a single jump instruction is inserted between the call and the target to maintain the semantics of the original code. [1] We want to *Stitch*(P_r, B) because the same considerations that applied to the previous if-then-else example apply here: if the frequent path through the procedure is small enough such that A, the body of P, and B could fit in the cache, then we would like to construct the thread shown on the right of Figure 5. The procedure *ThreadCall* effectively constructs the arc $P_r \rightarrow B$ such that P_r and B are eventually *Stitch*ed. [2]

3 Results

We wrote an instrumentation program that inserts code into the Gnu C compiler's assembly language output to collect arc frequencies at runtime. After

[1] We tried turning the *call* plus *jump* sequence into a ***push return address*** so A could fall through to P_0; the results were insufficiently interesting to warrant the additional complexity. Also, this would not work well on CISC machines that combine the *call* instruction with register saving.

[2] After the results for this paper were generated, we discovered that our implementation of this heuristic would occasionally incorrectly handle nested procedure calls. The net effect is that our numbers may show slightly less performance improvement than would be obtained if the bug were fixed.

	trace length			
name	unreorganized	reorg. $p = .80$	reorg. $p = .90$	reorg. $p = .95$
SCRUNCH	9,405,156	9,656,437	9,715,946	9,693,277
TROFF	8,059,174	8,343,440	8,325,470	8,338,350
CC1	8,263,593	8,268,313	8,293,376	8,301,226

Table 1: Summary of traces: number of instruction words fetched

	number of	number of blocks reorganized		
name	basic blocks	$p = .80$	$p = .90$	$p = .95$
SCRUNCH	1,233	22 (1.8%)	27 (2.2%)	32 (2.6%)
TROFF	4,000	149 (3.7%)	207 (5.2%)	318 (8.0%)
CC1	26,407	727 (2.8%)	1,317 (5.0%)	1,939 (7.3%)

Table 2: Summary of traces: number of basic blocks

profile data has been collected, our program *reorgBBs* then reads the original assembly language files and reorganizes them based on that profile.

3.1 The Programs and Traces

There were three programs chosen for our experiments and each program had four versions created: the normal, unreorganized version produced by the Gnu C compiler, and three reorganized by the algorithm in Figure 3 with p set to .80, .90, and .95. A summary of the programs, the basic block counts, and trace sizes is in Tables 1 and 2. We collected a trace of each of these twelve programs which we then used as input to Hill's DineroIII cache simulation program [15]. Each trace was simulated on eleven different cache configurations: 256 bytes with 4 and 8 byte blocks; 1024 byte cache with 4, 8, and 16 byte blocks; and 4096 byte cache with 4, 8, and 16 bytes, each using single associativity (direct mapped) and two-way set associativity.

Our first program was *scrunch*, a Huffman encoding algorithm. A profile was generated by *scrunch*ing a 200K spelling dictionary. A trace was created by *scrunch*ing *scrunch.c*, a 42Kb C source file.

A second program, *troff*, was chosen because of its wide use in UNIX environments. A profile was generated by *troff*ing three separate technical documents, chosen with some hope that they represented typical use of the program. The first document consisted of 103K bytes after being preprocessed by *tbl*, *eqn*, and *grn*, This included 1933 lines (32K bytes) of *troff* commands, the remainder being plain text. The other two documents totalled 228K bytes and contained 4004 lines (73K bytes) of preprocessed *troff* commands. A trace was created by *troff*ing a reduced version of the first document of length 7705 bytes, of which 273 lines (2728 bytes) were *troff* commands.

A third program was the Gnu C compiler itself. A profile was collected of the compiler compiling three Gnu C source files: *toplev.c*, *loop.c*, and *recog.c*. They totaled 79K bytes, with 20, 12, and 15 C function definitions, respectively. A trace was taken while compiling genemit.c, a 6Kb file containing nine function definitions.

3.2 Miss Rates

Figures 6, 8, and 10 are histograms of the results of running the traces of the programs through the cache simulator. (Since the reorganization algorithm didn't take any cache parameters into account, the same traces are used in all eleven simulation runs for each program.) They show the miss rates for the four versions of each program on each of the eleven cache configurations. The leftmost bar of each group of four is the average miss rate of the unreorganized program. The other three of the group, from left to right, are the average miss rates for the reorganized versions for $p = 80\%$, 90%, and 95% respectively. The cache configuration is noted beneath each group. Figures 7, 9, and 11 show the improvement in miss rates of the reorganized versions over the miss rates of the unreorganized versions (i.e. $1 - (M_r/R_r)/(M_u/R_u)$; see below).

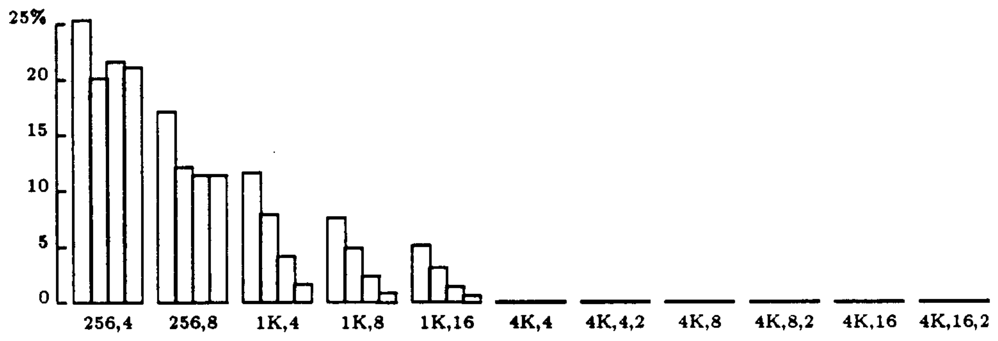


Figure 6: Scrunch miss rates

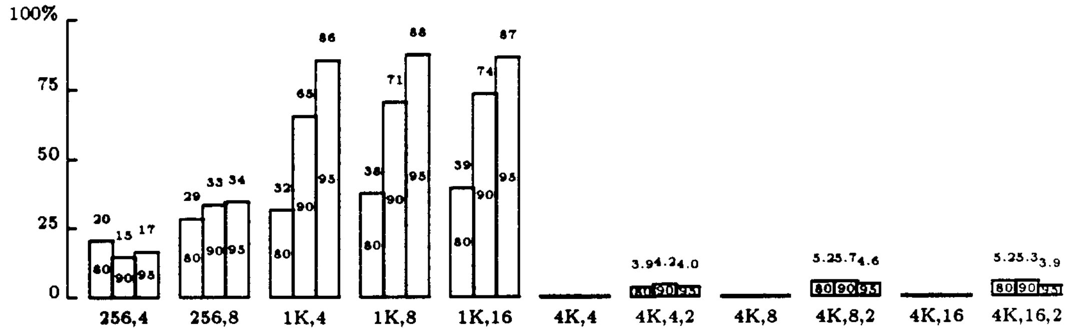


Figure 7: Percent improvement in scrunch's miss rate

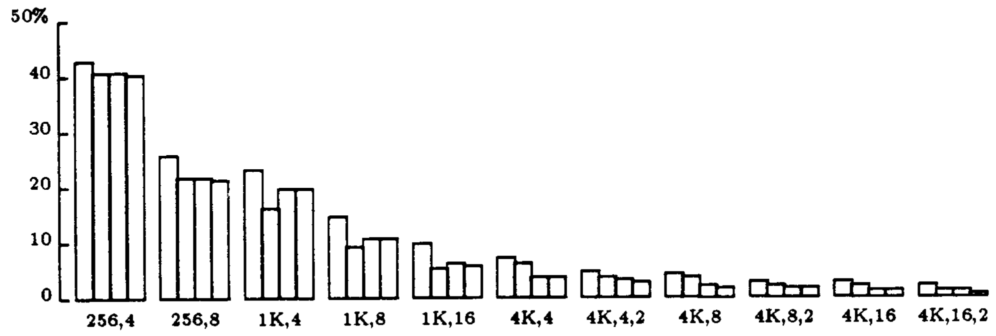


Figure 8: Troff miss rates

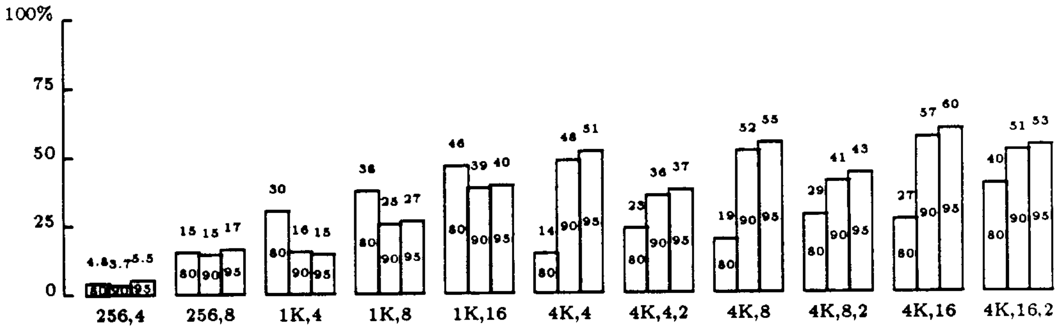


Figure 9: Percent improvement in troff's miss rate

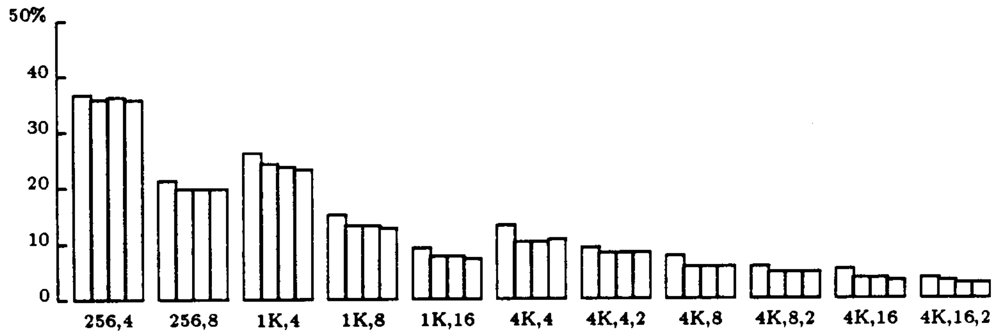


Figure 10: Ccl miss rates

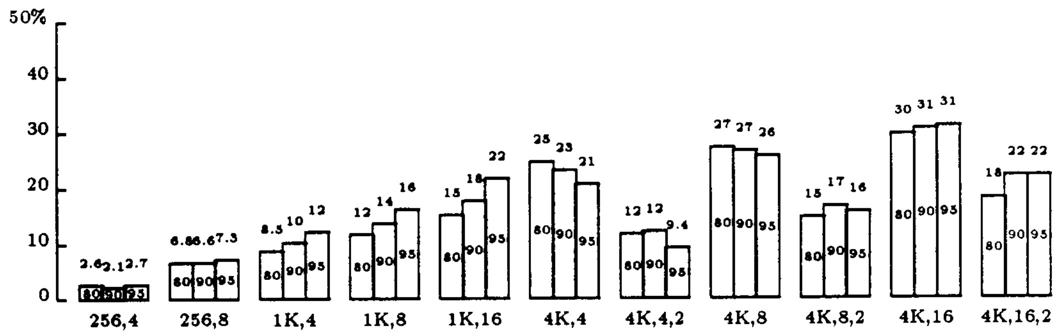


Figure 11: Percent improvement in cc1's miss rate

Note that there are instances where reorganization can buy the (miss rate) equivalent of a larger cache. For example, looking at Figures 6, 8, and 10 we see that the reorganized programs using a 256 byte cache with 8-byte blocks consistently had as good as or better miss rates than its unreorganized version running in a 1K cache with 4-byte blocks.

3.3 Performance improvement

What do these improvements in miss rates imply about the running time of the program? Let R_u be the number of instruction fetches on the original, unreorganized program, and let R_r be the corresponding number for a reorganized version. For the sake of these estimates, we will assume $R = R_u = R_r$. We see from Table 2 that this is not strictly true, but they are sufficiently close for our purposes here. Let M_u be the number of cache misses and H_u the number of hits when the original, unreorganized program is run, and M_r and H_r be the corresponding values for the reorganized program. Define the miss rates $m_u = M_u/R$ and $m_r = M_r/R$, and let $m_\Delta = m_u - m_r$ be the difference in miss rates. Let t_h be the time required to handle a cache hit, and let t_m be the time required to handle a cache miss. The running time of the original program is then $T_u = t_h h_u + t_m m_u$ and the improved running time is $T_r = t_h h_r + t_m m_r$. Finally, define $f = t_m m_\Delta / T_u$, the fraction of the original program's time taken up by cache misses that are turned into cache hits, and $K = t_m/t_h$ the ratio of the cost to handle a miss to the cost to handle a hit. Then remember Amdahl's Law:

$$T_r/T_u = (1 - f) + f/K. \qquad (1)$$

We can now estimate the improvements in performance from reorganization. We will take as our example, the SPUR memory architecture [14]. In general, the cost of a miss is very high on multi-processor, shared bus systems due to bus contention and/or the length of the cache line. SPUR has a 512-byte on-chip cache and 128Kb off-chip cache. A miss in the on-chip cache costs three times as much as a hit, assuming the instruction to be in the off-chip memory cache [16]. Let us assume the on-chip cache shows a normal miss rate of about 20%, and that we can improve that to 15% by reorganizing. Then $f = 3 * .05/(3 * .20 + .80) = .107$. Plugging this into (1) above, we get $T_r/T_u = .929$, i.e. the program executes in only 92.9% of the time of the original, a 7.1% improvement. The maximum possible improvement is 29.6% assuming the unattainable miss rate of 0%.

For the SPUR architecture, an off-chip cache miss will cost 12 to 20 times that for handling a cache hit. SPUR therefore has a very large mixed cache to combat this penalty. If we assume that reorganization can reduce SPUR's miss rate by an absolute 0.25% (e.g. from 1% to 0.75%), then, assuming $K = 17$ [17], $f = 17 * .0025/(17 * .01 + .99) = .0366$. Plugging this into (1) above, we get $T_r/T_u = .966$, a 3.4% improvement in performance. This is in addition to the performance improvement for the on-chip cache noted above. With these assumptions, we predict reorganization can improve SPUR's performance by about 10%.

Figures 12, 13, and 14 show the theoretical improvement in execution performance of the reorganized versions over the original unreorganized versions when the cost factor $K = 2, 3, 4, 5, 6, 7, 10, 15, 20$, and 25. Each graph has a column for each of the eleven cache configurations. A line within a column plots the expected performance of the indicated program on that cache when reorganized with the three values $p = .80, .90, .95$ moving from left to right. $K = 2$ is the very top line in each column, and $K = 25$ is the bottom-most line in each column, yielding a range in which we expect reorganization to improve the performance of the programs. So we see that for a 1K cache with 4-byte blocks and $K = 2$, a version of *scrunch* reorganized with $p = 80\%$ would take about 97% as long as the unreorganized version (the left end of the topmost line in the third column from the left). It would take only about 77% as long if $K = 25$ (the left end of the bottom line in that column), and only about 38%

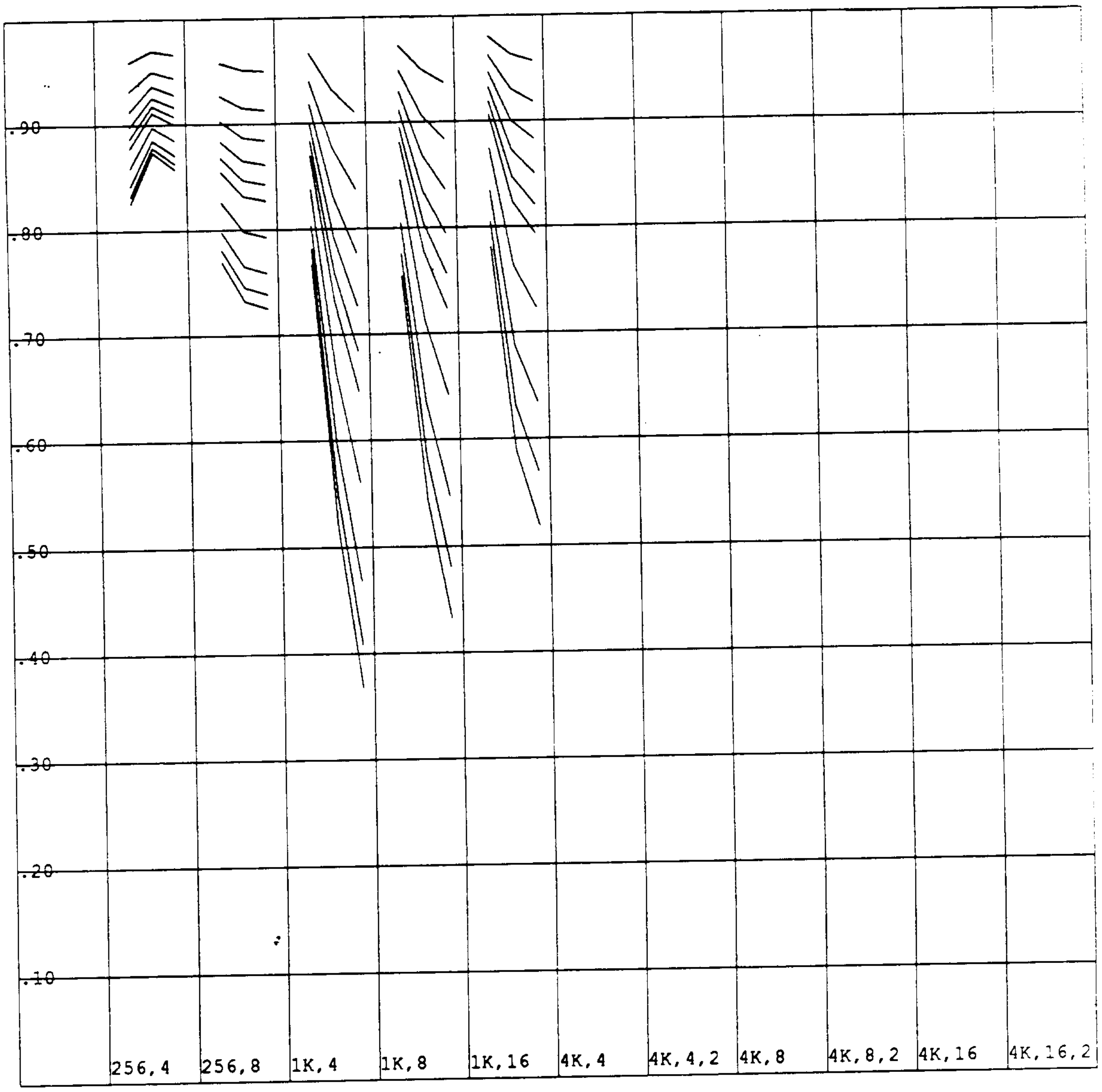


Figure 12: Scrunch improvements for K = 2, 3, 4, 5, 6, 7, 10, 15, 20, and 25

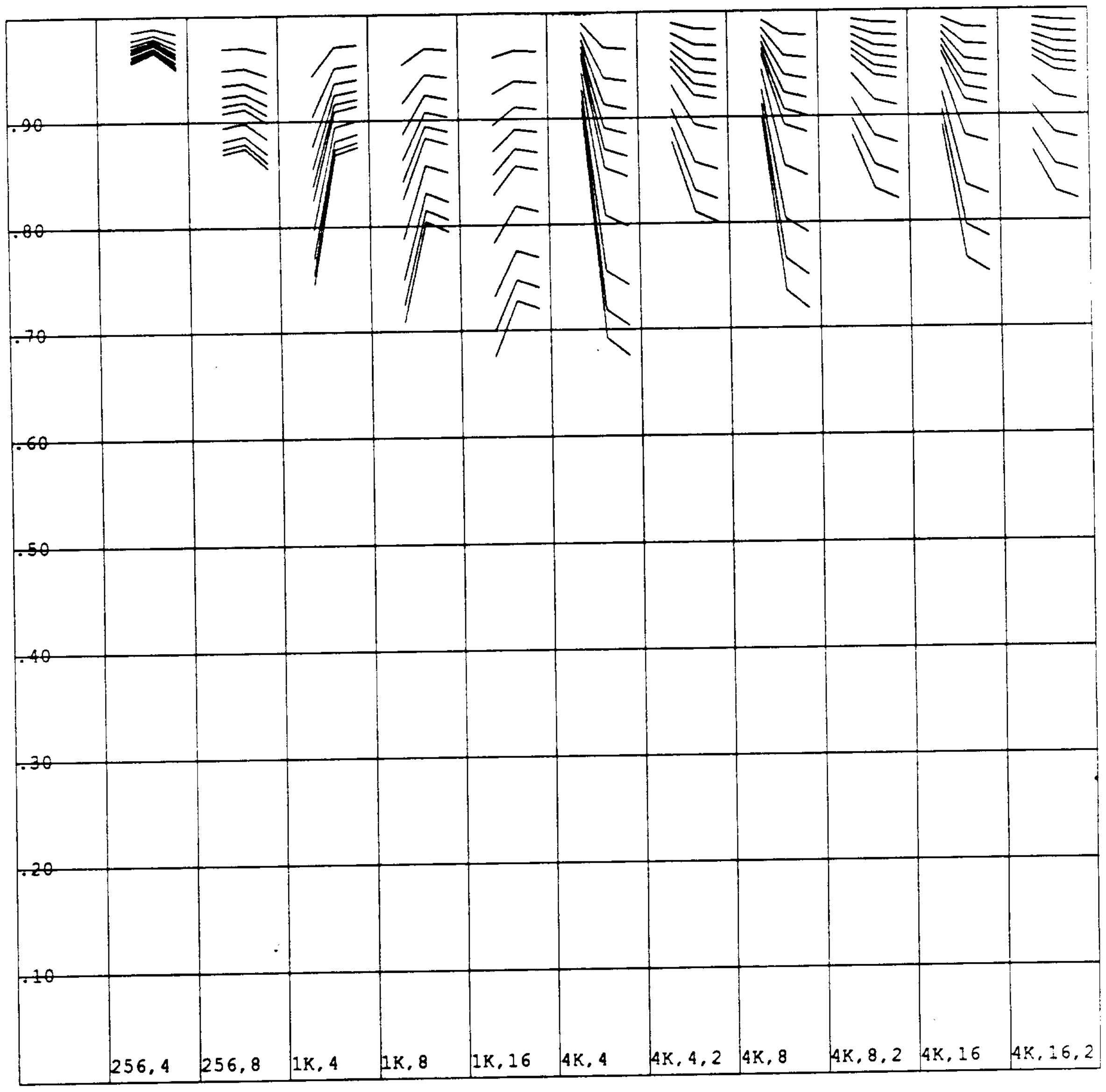


Figure 13: Troff improvements for K = 2, 3, 4, 5, 6, 7, 10, 15, 20, and 25

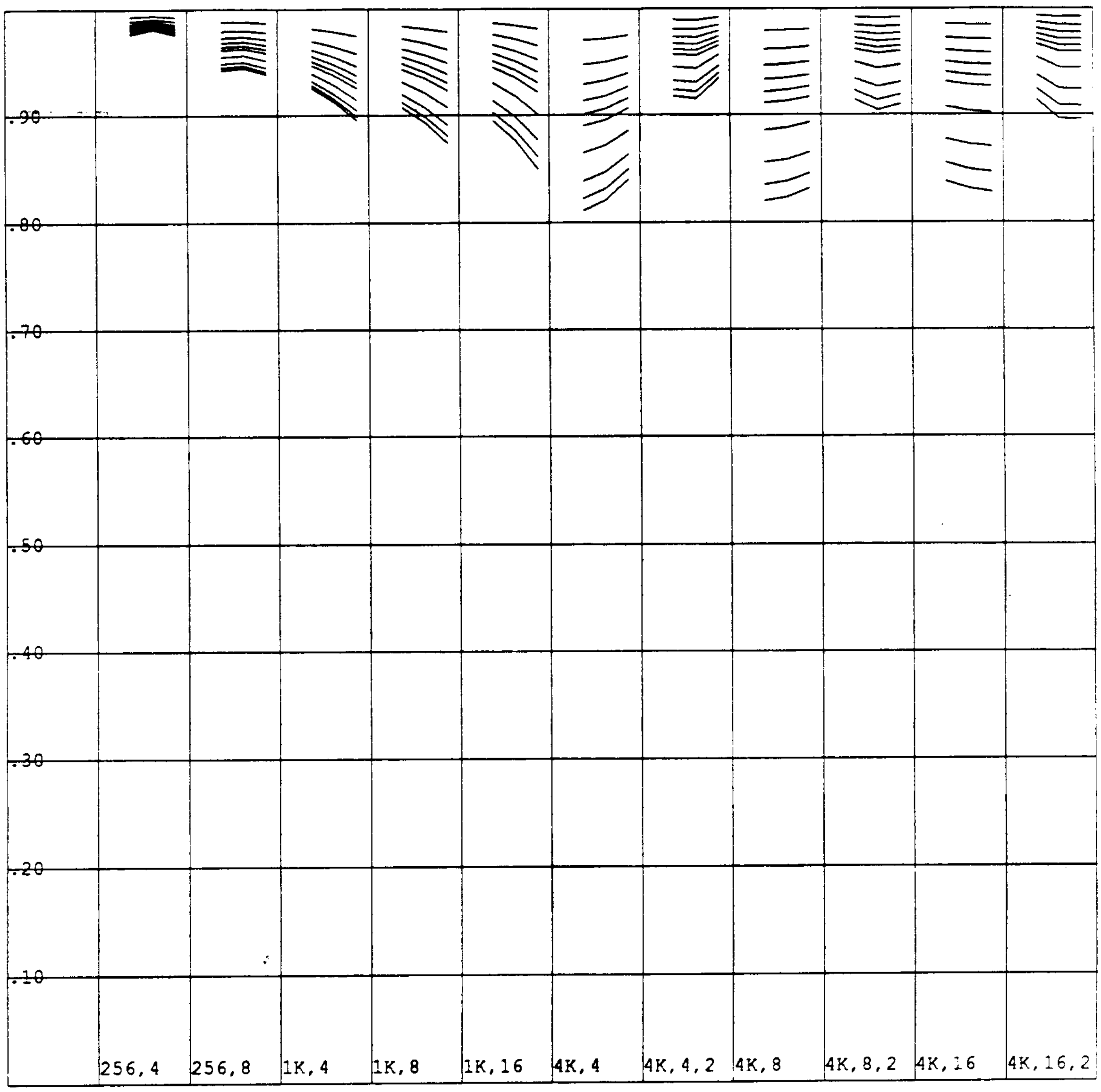
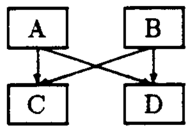


Figure 14: Cc1 improvements for K = 2, 3, 4, 5, 6, 7, 10, 15, 20, and 25

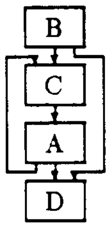
as long when $K = 25$ and $p = 95\%$ (the rightmost end of the bottom line in that column).

4 Arc Counts

We have chosen to use arc counts instead of simply counting the number of times each basic block is entered. Most profiling tools count the latter rather than the former. However, this is not quite what is desired, since it is too easy for basic block counts to mislead a reorganizer. For example, consider the following fragment of a program control flow graph:



There is no way, given only basic block counts, to recover the traversal counts for the arcs. Without the traversal arcs, there is no way of knowing whether, for example, D should follow A or B. By rearranging the picture and adding the arc $C \rightarrow A$, we see that this is not a rare construct. In fact, this construct appeared several times in code generated by the MIPS-X Pascal compiler for a program that solved boolean equations.



To know whether to stitch $B \rightarrow C$ or $B \rightarrow D$ requires knowing whether C_c comes from the loop or from B falling through to C. Basic block counts cannot make that distinction. Conversely, if the arc counts are available, basic block counts are easily derived by propagating frequencies along the arcs and through the basic blocks. The conclusion is that collecting only basic block counts is insufficient for reorganization.

Collecting arc counts is more expensive than simply counting basic block executions. In our implementation, the instrumented subject programs' execution times went up anywhere from 25% to 100% over the uninstrumented form, depending on the density of conditional branches. We did not attempt optimal instrumentation, although there are results in that area [19].

We do have an idea for a reasonably efficient method for profile collection that will give arc frequency counts, although some of them will only be approximate. Some profiling tools use the technique of interrupting the program at regular intervals and sampling the *pc* to derive a statistical estimate of the execution frequency of basic blocks. If the compiler located the (small) set of arcs whose instrumentation was necessary to recover all arc counts, i.e. just precisely those arc counts needed to resolve ambiguous instances like the one noted above, then that information could be combined with the statistical data from *pc* sampling to propagate estimates of execution frequency to all arcs and basic blocks. Furthermore, since this set of arcs is not a unique set, the compiler can actually use profile data from previous runs to determine which infrequently traversed arcs to instrument, thereby reducing the amount of instrumentation overhead. Graham et al. [12], note that the *pc* sampling technique adds about five to thirty percent overhead, so we can hope to increase that very little, if at all, and obtain arc counts at a more reasonable cost.

5 Limitations

We have not considered many of the parameters of engineering a cache that are obviously relevant for system design. For example, we have not considered the increase in bus contention caused by going to a larger block size. Nor have we considered alternative cache designs such as sub-block placement. Our goal was simply to explore improving the performance of the cheapest of these design alternatives with compiler modifications.

We have not considered cache effects such as cold start misses or cache flushes due to system interrupts or context switches. As noted above we did not consider any parameters of the target caches when reorganizing code. One reason for this elision was that it was not clear at the beginning how much the parameterization of the algorithms by the cache characteristics would benefit the program's performance; hence we went for the simple solution first.

We haven't solved the problem of *case* statements satisfactorily. Currently, it is possible for the reorganizer to move the code around to such an extent that the jump table can end up quite a distance away from one or more of its targets. On the 68020, this presents a practical problem since jump tables with half-word *pc*-relative entries are much faster than full word entries. If an item of a case is a "hot spot", it is very difficult to relocate it and still satisfy the distance constraint of jump tables. The only program that gave us real problems was the Gnu C compiler, for which we generated full-word jump table entries. This does not change any of our miss rate results significantly, but in real life, it would be an unacceptably slow implementation due to the slower execution of the table jump.

Finally, there are architectures which present difficulties for our algorithm. An example is the MIPS-X instruction set [6], which has non-orthogonal conditional branch instructions. Due to the nature of the MIPS-X pipeline, each conditional branch instruction is followed by two instructions that are fetched before the CPU has determined whether the branch will be taken. Each conditional branch instruction also has a *squash* bit that, if 'on', prevents the execution of these two instructions if the branch is not taken. Both of these delay slot instructions are always executed whenever the branch is taken: there is no way to ***squash*** their execution when the branch is taken. This makes sense if all programs follow the pattern of code generated by most compilers, where the conditional test is at the 'bottom' of the loop and the conditional branch is, therefore, almost always taken.

However, the reorganization we are describing here turns those statistics upside down: the greedy sewing algorithm almost guarantees that the head and tail of a frequently executed loop will be made contiguous, and that execution will almost always fall through the conditional test to the head of the loop: the conditional branch is almost always never taken. This leaves three options: (1) fill the delay slots as is currently done by MIPS-X compilers, followed by a jump instruction to the infrequent target (normally the loop-exit); (2) reverse the sense of the conditional and try to fill the delay slots with instructions that don't have to be squashed when the branch is *not* taken (because there is no squash bit for this direction); or, (3) punt and put no-ops in the delay slots.

Option (1) puts infrequently executed instructions right in the middle of high-frequency basic blocks, working against one of the aims of reorganization (better cache utilization in high frequency code). Option (2) sounds plausible, but it is difficult to find instructions that can always be executed no matter which way the branch goes. It may be possible to generate instructions to undo the effects of the delay-slot instructions when the branch is finally taken, but this begins to get complicated and presents the possibility of really slowing down a frequently executed inner loop. Option (3) is an obvious loss.

Due to the fact that the available compilers filled the delay slots before emitting assembly language code, we could not apply our algorithms to MIPS-X code. We cannot say precisely how badly our algorithms are hurt by the non-orthogonality of the conditional branch instructions. McFarling [21] implemented option (1), and reports that the size of repositioned code increases about 14%.

6 Conclusions

Profile driven code reorganization definitely improves the performance of programs. In envisioned programming environments where profile data is a permanent part of the information manipulated by both programmer and compiler, these improvements would come simply and cheaply. In the contemporary compile-a-file world, reorganization should be made available to programmers optimizing a working program for speed. Our experiments have shown improvements in miss rates on the order of 30% to 50%, and sometimes as high as 50% to 80%. These figures were obtained by relocating only 3% to 8% of the basic blocks of typical programs.

1. Babonneau, J. Y., Achard, M. S., Morisset, G. and Mounajjed, M. B. Automatic and General Solution to the Adaptation of Programs in a Paging Environment. *Proceedings of the 6th Symposium on Operating System Principles*, West Lafayette (November 1977).

2. Baer, J-L. and Caughey, R. Segmentation and Optimization of Programs from Cyclic Structure Analysis. *Proceedings AFIPS Spring Joint Computer Conference*, *40* (1972), 23–35.

3. Baer, J-L. and Sager, G. R. Dynamic Improvement of Locality in Virtual Memory Systems. *IEEE Transactions on Software Engineering*, *SE-2*, 1 (March 1976), 54–62.

4. Barrese, A. L. and Shapiro, S. D. Structuring Programs for Efficient Operation in Virtual Memory Systems. *IEEE Transactions on Software Engineering* (November 1979).

5. Chanson, S. T. and Law, B. Strategy-independent Program Restructuring Based On Bounded Locality Intervals. Department of Computer Science University of British Columbia, 81-9, 1981.

6. Chow, P. MIPS-X Instruction Set and Programmer's Manual. CSLSU, CSL-86-289, May 1986.

7. Denning, P. J. Virtual Memory. *ACM Computing Surveys*, *2*, 3 (September 1970), 153–189.

8. Fabri, J. *Automatic Storage Optimization*. PhD Dissertation, Courant Institute of Mathematical Sciences, NYU, 1979.

9. Ferrari, D. Improving Program Locality by Strategy-Oriented Restructuring. *Information Processing, Proceedings IFIP Congress 74*, New York, NY, Amsterdam (1974).

10. Ferrari, D. Program restructuring algorithms for global LRU environments. *Proceedings International Computer Symposium*, Liege, Belgium (April 1977).

11. Ferrari, D. and Lau, E. J. An Experiment in Program Restructuring for Performance Enhancement. *Proceedings 2nd International Conference on Software Engineering*, San Francisco, Calif. (October 1976).

12. Graham, S. L., Kessler, P. B. and McKusick, M. K. gprof: A Call Graph Execution Profiler. *Proceedings of the ACM-SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices*, *17*, 6 (June 1982), 120–126.

13. Hatfield, D. J. and Gerald, J. Program Restructuring for Virtual Memory. *IBM Systems Journal*, *10*, 3 (1971), 168–192.

14. Hill, M. D., Eggers, S. J., Larus, J. R., Taylor, G. S., Adams, G., Bose, B. K., Gibson, G. A., Hansen, P. M., Keller, J., Kong, S. I., Lee, C. G., Lee, D., Pendleton, J. M., Ritchie, S. A., Wood, D. A., Zorn, B. G., Hilfinger, P. N., Hodges, D., Katz, R. H., Ousterhout, J. and Patterson, D. A. Design Decisions in SPUR. *IEEE Computer*, *19*, 11 (November 1986).

15. Hill, M. D. DineroIII Cache Simulator. University of California, Berkeley, UNIX Programmer's Manual, August 1985.

16. Hill, M. D. Aspects of Cache Memory and Instruction Buffer Performance. Computer Science Division, EECS, University of California, Berkeley, Technical Report UCB/CSD 87/381, PhD Dissertation, November 1987.

17. Katz, R. H., Eggers, S. J., Gibson, G. A., Hansen, P. M., Hill, M. D., Pendleton, J. M., Ritchie, S. A., Taylor, G. S., Wood, D. A. and Patterson, D. A. Memory Hierarchy Aspects of a Multiprocessor RISC: Cache and Bus Analyses. Computer Science Division, EECS, University of California, Berkeley, UCB/CSD 85/221, January 1985.

18. Knuth, D. E. An Empirical Study of FORTRAN Programs. *Software–Practice & Experience*, *1* (1971), 105–133.

19. Knuth, D. E. and Stevenson, F. R. Optimal measurement points for program frequency counts. *BIT*, *13* (1973), 313–322.

20. Lau, E. J. *Performance Improvement of Virtual Memory Sytems.* UMI Research Press, Ann Arbor, MI Research Press, 1982.

21. McFarling, S. Program Optimization for Instruction Caches. *Symposium on Architectural Support for Programming Languages and Operating Systems*, Boston, MA (April 3-6, 1989), (to be published).

22. Paris, J-F. Application of Restructuring Techniques to the Optimization of Program Behavior in Virtual Memory Systems. PhD Dissertation, University of California, Berkeley, PROGRES report 81.5, May 1981.

23. Ryder, K. D. Optimizing Program Placement in Virtual Systems. *IBM Systems Journal*, *13* (1974), 292.

24. Thabit, K. O. *Cache management by the compiler.* PhD Dissertation, Rice University, November 1981.