

Profile Guided Code Positioning

Karl Pettis and Robert C. Hansen

Hewlett-Packard Company
California Language Laboratory
19447 Pruneridge Avenue
Cupertino, California 95014

Abstract

This paper presents the results of our investigation of code positioning techniques using execution profile data as input into the compilation process. The primary objective of the positioning is to reduce the overhead of the instruction memory hierarchy.

After initial investigation in the literature, we decided to implement two prototypes for the Hewlett-Packard Precision Architecture (PA-RISC). The first, built on top of the linker, positions code based on whole procedures. This prototype has the ability to move procedures into an order that is determined by a "closest is best" strategy.

The second prototype, built on top of an existing optimizer package, positions code based on basic blocks within procedures. Groups of basic blocks that would be better as straight-line sequences are identified as *chains*. These chains are then ordered according to branch heuristics. Code that is never executed during the data collection runs can be physically separated from the primary code of a procedure by a technique we devised called *procedure splitting*.

The algorithms we implemented are described through examples in this paper. The performance improvements from our work are also summarized in various tables and charts.

1. Introduction

Traditional optimization techniques attempt to improve program performance by eliminating instructions, both statically and dynamically. These traditional techniques fall short of achieving optimal performance for today's architectures because they do not directly address improving the performance of the instruction memory hierarchy. The instruction memory hierarchy has increasingly become more of a performance bottleneck because:

- Processor speeds continue to increase at a much faster rate than memory speeds.
- For RISC type architectures, there is a two fold increase in instruction memory requirements as compared to CISC type architectures [DV87].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1990 ACM 0-89791-364-7/90/0006/0016 \$1.50

Proceedings of the ACM SIGPLAN'90 Conference on
Programming Language Design and Implementation.
White Plains, New York, June 20-22, 1990.

Investigations by engineers within Hewlett-Packard have supplied evidence that the instruction memory hierarchy for PA-RISC is a potential performance bottleneck. For many applications, poor program locality may lead to:

- Higher instruction cache and TLB (Translation Lookaside Buffer) miss rates. The first implementations of PA-RISC showed a CPI (Cycles Per Instruction) of about 3 for the MPE/XL† operating system. Of this, 1 of the 3 cycles was due wholly to instruction cache misses. Subsequent hardware implementations have improved, but cache misses still account for a large portion of the CPI.
- Excessive page thrashing even under small workloads due to working sets being too large.

There have also been earlier experiments within Hewlett-Packard attempting to improve program locality with very favorable results. Most notable are the following:

- A prototype Pascal compiler, built by moving frequently executed basic blocks to the top of procedures, had a 40 percent reduction in the number of instruction cache misses and an overall performance improvement of 5 percent.
- A prototype Fortran compiler, built by hand arranging the object files to produce better procedure call locality, received a 20 percent system throughput improvement for an internal benchmark simulating a Fortran multi-user/application development environment.

Given this, we set out to develop prototype compilers and linkers that would use execution profile information as feedback to guide in the positioning of code in order to reduce the overhead of the instruction memory hierarchy. Specifically, we attempt to improve the performance of the instruction cache, although additional benefits are realized.

Our approach to position code at the procedure level is primarily implemented in a modified linker, while the basic block positioning was added to a traditional optimizer in our compilers. We also implemented a technique of *procedure splitting* that separates procedures into two parts, yielding additional benefits. This was also implemented in the optimizer.

The specific implementation of these techniques was done for PA-RISC. Although there is some specific tuning in the basic

† MPE/XL is a proprietary commercial (OLTP) operating system for the PA-RISC architecture.

block positioning for the PA-RISC branch prediction algorithm, the techniques and ideas we developed are generally machine independent and applicable to many architectures.

2. Related Work

The problem of restructuring programs to improve memory performance has been studied by numerous researchers. Most of the earlier work concentrated on reducing page faults for virtual memory machines. Hatfield and Gerald [HG71] used a graph clustering algorithm to group routines that executed spatially together onto the same page. Ferrari [Fer74] [Fer76] defined a similar algorithm modeling how pages would be replaced based on the working set model of a program. Hartley [Har88] used duplication of code modules and inlining to extend the concepts of page locality.

More recently, algorithms for positioning code at the basic block level have been presented. McFarling [McF89] uses simple profile data to guide in excluding some instructions from the cache to significantly increase the performance of a direct mapped cache. Hwu and Chang [HC89] also use profile data to guide in the positioning of basic blocks.

Our approach with basic blocks is most similar to Hwu and Chang, although our algorithms are significantly different since the development was done independently. We also differ in implementation strategy. Hwu and Chang use a preprocessor to modify the source (C only) to do instrumentation and then use the profile data to position code generated by their C compiler. Because we did our basic block instrumentation and positioning work in an optimizer shared among many compilers, we are able to use our techniques on large multi-language and multi-source file applications.

3. Prototype 1: Procedure Positioning

The first prototype developed uses dynamic call graph information to guide in the positioning of procedures. We adopted a “closest is best” strategy. In other words, if a procedure calls another frequently, we want the two procedures to wind up close to one another in the final code image. Doing this for two procedures increases the chances that they will land on the same page, thus reducing the size of the page working set. Minimizing the page working set should result in fewer page and TLB misses. But more importantly, since the instruction cache for a single process on PA-RISC (as well as many other architectures) acts in a *modulo* fashion, placing two procedures next to one another minimizes the overlap in cache lines between them. This should result in fewer cache collisions implying fewer cache misses.

Another benefit of “closest is best” is reducing the number of executed *long branches*. On PA-RISC, the simple short branch instruction used for procedure calls has a range of $\pm 64K$ instructions. By software convention, the compilers normally assume that the simple branch instruction is sufficient. It is the linker’s responsibility to insert a *long branch stub* if the source and target of a procedure call are separated by a greater distance. This stub nominally takes three cycles to execute; however, there often will be an additional cache miss for the stub plus an extra penalty associated with branch prediction.

From our “closest is best” strategy, we dramatically reduce the number of these long branches executed dynamically, even though we often saw an increase in the static count for the number of stubs required. Refer to the *Results* section for additional detail.

3.1. Measurement Method

In our first effort for procedure positioning, we chose to use *gprof* [GKM82] profiling data as input to the positioning algorithm. This was done since *gprof* was a supported measurement method that we could use very quickly. In particular, we built a call graph weighted by the procedure call counts using the profile information obtained from *gprof*. We then wrote a separate utility to manipulate the graph to determine a link order for the procedures.

We found that using *gprof* had some drawbacks that caused us to change to measurement done by a modified linker:

- *Gprof* requires recompilation to add measurement support.
- We wanted to be able to measure library routines. Without access to source code for library routines, we could not compile with *gprof* support.
- Assembly routines are not handled by *gprof*.
- The linker provides a notion of *subspaces* within the code area. It was deemed impractical for the first prototype to move code in anything less than subspace size chunks. At the time we did the prototype, all procedures within a relocatable file were included into a single subspace. But *gprof* measures based on individual procedures, not subspaces. Therefore, if two or more procedures within the same relocatable file needed to be placed in different positions according to our algorithm, we had to make an arbitrary choice.

This problem has been eliminated by new *procedure-per-subspace* compilers (each procedure within a relocatable file is contained in a separate subspace), but we did not receive such compilers until quite late in our development of the first prototype.

Looking for a replacement for *gprof* to solve the above problems, we modified the linker to insert measurement code.

The linker sees all direct calls between the various code subspaces. For each such call from one subspace to another, the linker was modified to drop down a stub between the caller and callee to count the number of control transfers.

The counters are maintained in a data space of the application. The linker initializes the counter area to zeroes and makes it big enough to handle all of the counters needed. During execution of the application, the individual counters in the data area are updated as calls pass through the linker added stubs. At the end of execution, the values of the counters are written to a file.

For this prototype, indirect calls using procedure pointers were not measured.

3.2. Procedure Ordering

The procedure ordering algorithm is implemented in a separate tool from the linker. The first step in its ordering of procedures is to construct an undirected weighted call graph using the collected profiling data. Initially, each node of the graph is a single procedure and the edges correspond to calls between the procedures with the edges weighted by the number of times the calls were actually made.

If a procedure calls another from several different places, or if two procedures are mutually recursive, those weights are first merged together in a single edge in the graph. The next step is to begin building the link order for the procedures in a bottom-up method using our “closest is best” algorithm.

The algorithm first chooses the edge with the heaviest weight. If multiple edges in the graph have identical weights, one is arbitrarily chosen. The two nodes connected by the chosen edge will be placed next to each other in the final link order. Then in the graph, the two nodes corresponding to this edge are merged into one and the remaining edges leaving each node are coalesced. This process continues until the whole graph consists of one or more individual nodes with no edges.

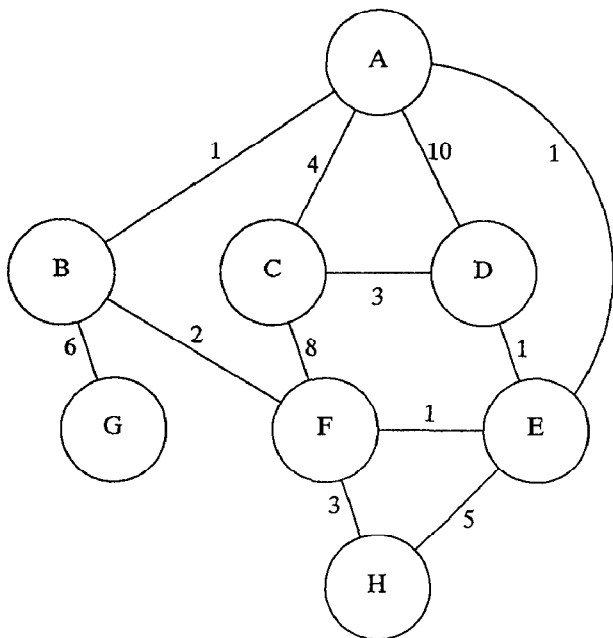


Figure 1: Original Graph

For example, consider the call graph in Figure 1. The heaviest edge (weight 10) is the one connecting procedures A and D.

So we decide that procedures A and D will be next to one another in the final link order, although which will actually be first is not decided at this point. We merge their two nodes in the graph and then repeat the process. The next edge chosen is the one between procedures C and F (weight 8). Again, although we have determined that these will be next to one another in the final link, we do not yet know which will be first. After merging nodes C and F, we have the situation shown in Figure 2.

The next heaviest edge (weight 7) is the edge that connects the two nodes that we have just built by our previous mergers. Consider the four procedures making up these nodes. Since we have already decided that A and D should be together and that C and F should be together, there are four distinct choices available for the ordering of the next merger (since at this point we consider an ordering to be interchangeable with its reverse ordering). The four choices are:

A-D-C-F or F-C-D-A
 A-D-F-C or C-F-D-A
 D-A-C-F or F-C-A-D
 D-A-F-C or C-F-A-D

To determine which of these to choose by our "closest is best" strategy, we look at the original connections between the procedures. We find that procedure F is not connected to either A or D and procedure C is more strongly connected to

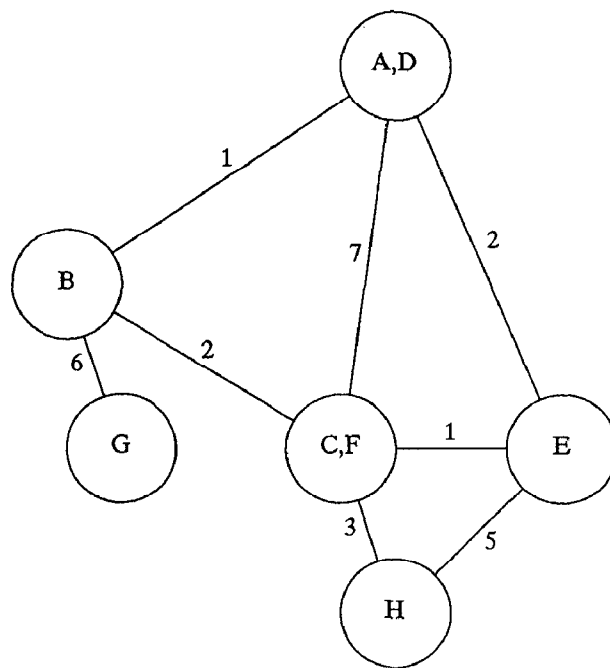


Figure 2: After Second Merger

A (original weight 4) than D (original weight 3). So we prefer that procedures C and A be adjacent in the final order, as long as the already determined groupings can be satisfied. Therefore, the order we choose for the four procedures is D-A-C-F (or the reverse ordering, F-C-A-D).

This process continues until the graph has no edges left. If the graph was originally a connected graph, then the final step will lead to a single node. If it was a disjoint graph, then there will be several independent nodes. This can happen if some procedures were not called at all (caller/callee arcs never taken are not added to the graph), or if the mechanism for calling a procedure is not known to the prototype. For example, neither invocations through signal handlers nor dynamic calls are measured by the prototype. For the prototype, we decided that the effort required to fix these minor anomalies was not worth it.

The final step in the ordering process is to print the ordered list of procedures to a file to be used by the linker.

3.3. Procedure Placement

The linker already knew how to sort subspaces based on various attributes including *sort keys* and access rights. By relying on the *procedure-per-subspace* compilers to separate procedures into distinct subspaces, we found that we could place procedures in a specific order just by modifying the linker's sorting algorithm. The output of the graph ordering algorithm described in the previous section is a list of procedures in a linear order. We supply the linker with that list, which it uses in its sort of subspaces.

By using the existing linker machinery in this way, we found that we had to modify very little code.

The benefits that were achieved using the procedure positioning prototype are described in our *Results* section.

4. Prototype 2: Basic Block Positioning

The second prototype we developed uses profiling information to guide the positioning of basic blocks.

Within most applications there are control flow paths and basic blocks that are seldomly executed during typical runs. Given the way that traditional compilers generate code, these seldomly used sections of code are interspersed with frequently used sections. For example, consider the code for an error check:

```
if (test for error) then
    handle unusual case
more code
```

When this code is compiled, you typically have the code for the error test terminating a basic block with a conditional branch around the code to handle the unusual situation. In the normal case where there is no error, the branch will usually be taken.

This often means that not all of the instructions in the cache line containing the branch instruction, or the cache line containing the target, will be executed. Incompletely using some cache lines implies that the total number of cache lines used is higher than one might expect from the number of instructions actually executed.

In addition, most implementations of PA-RISC predict that forward conditional branches will not be taken and that backward conditional branches will be taken. For the code generated in the above example, the prediction for the forward branch is usually incorrect. This leads to wasted cycles due to processor stalls, a penalty which varies with different implementations of the architecture. Higher-end machines with a deeper pipeline have a larger penalty associated with an incorrect prediction.

What we want to do is to identify such cases and move the infrequently executed code away so that the normal flow of control is in a straight-line sequence. The target and sense of the conditional branch will be changed to reflect the new ordering of code. The benefits we expect from doing this are:

- Longer sequences of code are executed before taking a branch.
- On average, the number of instructions executed per cache line increases.
- There are fewer cache misses due to a denser instruction stream.
- Since the hardware prediction matches the normal execution, branch penalties are reduced.
- Better use of branch delay slots.
- In some cases (e.g. an *if-then-else* with a seldomly executed *else* clause), moving the infrequently executed code away results in the elimination of an unconditional branch.

Currently, the support for basic block profiling and the actual positioning are implemented within the early phases of the optimizer component of our compilers. This results in a two-pass approach for the user. The user first compiles with measurement support and then after collecting profiling data, recompiles for positioning.

4.1. Measurement Method

The first consideration we had in measurement was deciding what exactly to measure. Typical path coverage tools (e.g.

PFA†) measure the number of times each basic block is executed without regard as to how you reached the basic block. An alternative approach is to count the number of times control transfers from one basic block to another. Using a basic block control graph, the difference is that PFA measures the number of times each node of the graph is executed, while the second method measures the number of times each arc is traversed.

The measurement of the arcs provides more data than the measurement of nodes. It is easy to derive node measurements from arc measurements but the converse derivation is not generally possible.

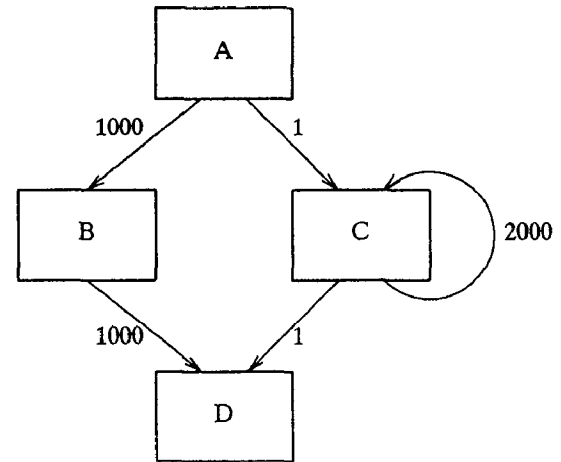


Figure 3: Arc vs. Node Measurement

Figure 3 shows a basic block structure with the arcs labeled with their execution counts. The PFA measurement for basic block C is 2001, while for basic block B it is only 1000. Therefore, an algorithm based only on PFA counts would be inclined to position basic block C after A, rather than basic block B. An algorithm based on arc measurements would be inclined to the reverse decision, since the arc from A to C is only executed once while the arc from A to B is executed 1000 times. Because of these considerations, we chose to implement measurement of arcs.

It was our original belief that implementing measurement of arcs would be significantly more complex than PFA-like measurement. It turned out, however, that it was relatively easy given that we decided to insert measurement code just before final optimization. The measurement code is inserted into the procedure after the basic block control graph has been built. Methods exist to reduce the number of distinct counters used [Sar89], since in many cases the counter value of some arcs is derivable using the flow graph and the values of other arcs. For our prototype, we did not implement these techniques. Instead, we generate code to increment a counter in the data area for every arc of the basic block graph.

In order to handle multiple compilation units during positioning, there needs to be a way to associate a group of counters with a particular compilation unit. We did this by placing a descriptor containing the relocatable file name, fully qualified, just before the counters in the data area. When the counters are dumped to a file after each profiling run, the file names

† PFA is a Path Flow Analyzer for PA-RISC.

are also dumped. When a file is compiled for positioning, the counters are scanned using the file name to find the group of counters corresponding to the procedures in that file.

4.2. Basic Block Ordering

While analyzing the weighted basic block control graphs for various examples, we developed two straightforward algorithms, *algo1* and *algo2*, to guide the basic block positioning. The principal difference between the two is that *algo1* is a top-down, depth-first algorithm while *algo2* is a bottom-up algorithm. Both methods showed significant improvements over traditional optimization techniques when applied by hand to small examples. For our prototype implementation, we chose to implement *algo2* because it performed marginally better than *algo1* in the examples we considered.

4.2.1. Algo2: Bottom-up Positioning

The idea in this algorithm is to form *chains* of basic blocks that should be placed as a group as straight-line code. These chains are built in a bottom-up method by initially considering each basic block as the head and tail of a chain by itself. Looking at the arcs of the basic block graph from largest to smallest, two different chains are merged together if the arc connects the tail of one chain to the head of another. In this case, the target chain is appended to the source chain.

If the source of the arc is not a tail or the target of the arc is not a head, then the chains cannot be merged. For example, if the more frequently executed arc out of a conditional branch results in the merger of chains (as it usually will), then when the less frequently executed arc is considered, no merger will be possible. The arc's source (the conditional branch) will already be in the middle of a chain.

Nevertheless, we do use these arcs to define a precedence relation among the chains to guide in the final positioning. Since the lesser arc will involve a branch, we want it to be a forward branch to match the hardware prediction that forward branches are less frequently taken. Therefore, the source of the arc should occur before the target. This implies that the chain containing the source is given precedence over the chain containing the target.

After all arcs have been looked at, there are a set of chains and precedence relations that will be used to do final placement. The algorithm starts with the entry chain, then chooses the next chain based on how heavy the inter-chain connections are, subject to the precedence relations defined by the conditional branches.

The precedence relation between any two chains is not necessarily transitive; it is possible for there to be loops in the precedence relations. If there is a loop, a decision is made based on the highest execution counts between the chains in conflict.

The principal idea behind this algorithm is to try to reduce the number of taken branches (conditional and unconditional).

4.2.2. Algo1: Top-down Positioning

The first observation for this algorithm is that basic blocks, except for branch table basic blocks, either have one or two successors. The algorithm starts by first placing the entry basic block for the procedure. Thereafter, the successor that is connected to the last placed basic block by the largest count is selected, as long as it has not already been placed. If all successors have already been selected, you pick among the unselected basic blocks the one with the largest connection to

the already selected blocks. This continues until all basic blocks are placed.

The principal idea behind this algorithm is to try to make conditional branches become forward conditionals that are usually not taken.

4.2.3. An Example

Figure 4 shows the weighted basic block graph of a procedure with each arc labeled with its profile count. The C source code was a *while* loop. Within the body of the loop are several *if-then-else* tests along with two alternate *break* statements (at basic blocks G and M).

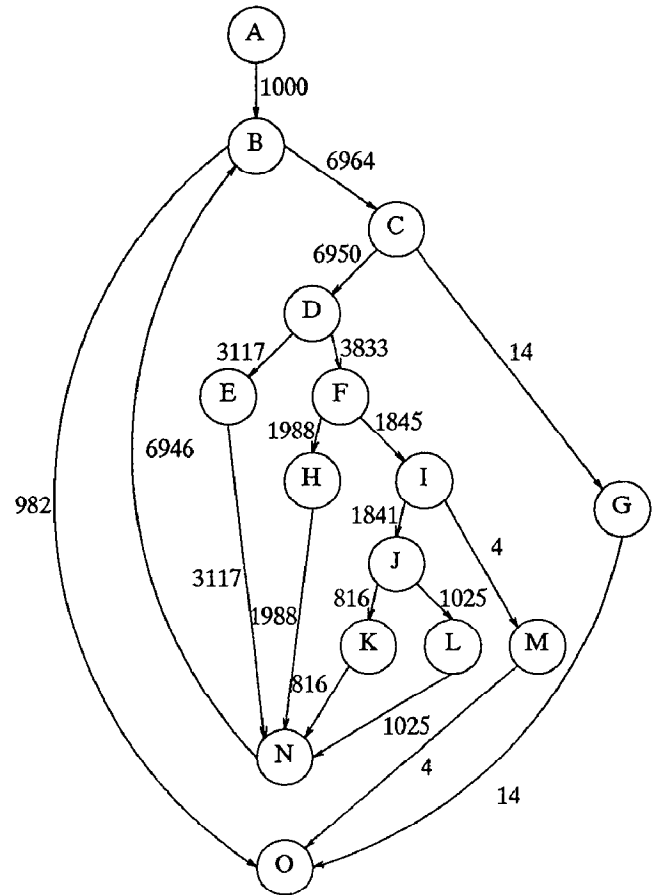


Figure 4: Basic Block Example

We start *algo2* by looking at the arc with the heaviest weight (6964) from basic block B to C and joining these two together into a chain. We then look at the next largest arc (6950) from basic block C to D. Since basic block C is at the end of a chain, we can extend the B-C chain to include D. The next three largest arcs (N-B, D-F, and E-N) also extend the existing chain. The next largest arc (D-E) can not be used to extend a chain, since basic block D is already in the middle of a chain. So we continue with the remainder of the arcs. The final result will be six chains (three are individual basic blocks):

- | | |
|------------------|--------|
| 1) A | 4) G-O |
| 2) E-N-B-C-D-F-H | 5) K |
| 3) I-J-L | 6) M |

After the chains have been formed, we then establish a precedence relation between them based on our desire to have

non-taken conditional branches be forward. There are 6 conditional branches in this procedure. By our criteria, the one at B indicates that chain 2 containing B-C should occur before chain 4 containing O. If these two chains are placed in this order, then the less frequently executed path out of B is the forward branch as desired.

The other conditional branches at C, F, I and J imply that you would like the following order among the chains:

chain 2 (C)	<i>before</i>	chain 4 (G)
chain 2 (F)	<i>before</i>	chain 3 (I)
chain 3 (I)	<i>before</i>	chain 6 (M)
chain 3 (J)	<i>before</i>	chain 5 (K)

The only conditional branch left at D shows that it is sometimes impossible to do a perfect job. By our criteria we want the branch from D to E to be forward (since that link is not taken as often as the path from D to F). But since E and D are part of the same chain, and E occurs before D in the chain, that branch is necessarily backwards. In other examples, it sometimes happens that the precedence relation between chains is not transitive. We resolve such conflicts by looking at the weights of the arcs and choosing an order that will satisfy the precedence of the larger arcs.

After we determine the precedence criteria between chains, we arrange them so that the order we want is satisfied as best as possible. If there is some freedom in choosing the next chain, we try to choose the chain connected to existing chains by the heaviest count. In this example, the final order of the basic blocks that we determine is:

A, E-N-B-C-D-F-H, I-J-L, G-O, K, M

Note that this ordering has altered the layout of the loop. In particular, the main test for the loop (basic block B) is now in the middle of the loop.

Algo1 (which we did not actually implement in the prototype) starts with the entry point basic block. At each point, we try to place a successor of the last placed basic block. If there is more than one available, we choose the one with the higher arc count. In this example, we would start with basic block A. It only has one successor, B, so that would be placed next. B has two successors, but C is connected with the higher arc count so it would be chosen next. Continuing in this way, we would place D, F, H and N.

After N is chosen, its only successor has already been selected, so we choose a new place to start from the remaining unselected basic blocks. We pick up with E since it is connected to the already chosen blocks by a weight of 3117. Note that its only successor has also already been chosen, so we start again with another unselected block, I. The final order after the algorithm completes is A, B, C, D, F, H, N, E, I, J, L, O, K, G, and M.

Both *algo1* and *algo2* do a good job of making conditional branches into forward branches that are usually not taken. *Algo2's* advantage seems to be that it does a better job at getting rid of unconditional branches.

4.3. Basic Block Placement

After determining the set of chains and their ordering, the final phase is to restructure the basic block graph before passing into the actual optimizer. Because internally the basic block graph is built using a doubly linked list of instructions, what really must be done is to properly break and reconnect

the instructions at the basic block boundaries. By and large, this process just involves pushing pointers around.

We start by placing the chains down, reconnecting the basic blocks to each other by means of the double links. Then, a pass is made over the basic blocks to insert any necessary branches to ensure that the code executes correctly. These branches are inserted where the fall-through after a basic block is no longer correct. For example, if the *then* clause of an *if-then-else* statement was moved away, the *else* basic block would immediately follow the branch and it would be in the fall-through location; however, the correct fall-through for the branch is the *then* clause. For correctness, a new basic block with a branch to the original fall-through location is inserted to be the fall-through.

This means that the conditional branch in the *if* clause is a branch around an unconditional branch. We could take the trouble to reverse the sense of the conditional branch and alter the target rather than just inserting a new basic block, but the optimizer already knows how to do this so we let it do the work later.

At this point, the procedure is ready to be passed through the optimizer.

5. Procedure Splitting

After a new ordering of the basic blocks has been established for a given procedure, the frequently executed basic blocks (*primary*) are now found toward the top of the procedure while the infrequently executed blocks are found near the end. Basic blocks that were never executed according to the collected profiling data (*fluff*) end up at the very end of the procedure.

Procedure Splitting is the process of separating the fluff basic blocks of a procedure into a separate region in an attempt to minimize the size of the primary procedure. The benefits of procedure splitting magnify the concept of locality. By producing smaller and denser primary procedures, more procedures can now be packed onto a single page. This should result in a further reduction of the page working set size and the number of page and TLB misses.

The left half of Figure 5 shows a simple example of three procedures that would normally require two pages of memory. By splitting the primary portions away from the fluff, the primary code for the procedures can be contained within a single page as shown in the right half of Figure 5.

When separating the fluff basic blocks from the primary blocks for a given procedure, we define a new "procedure" to encompass the fluff blocks. This procedure is not a typical one in that it does not adhere to the standard procedure calling convention: there is no defined entry or exit point and no register saves or restores. We do this to avoid the overhead associated with a standard procedure call whenever control transfers between a primary and fluff procedure. The new procedure is also given a lower sort key so that the linker will force all fluff procedures to the end of the code area.

After we determine which basic blocks will be moved into the fluff procedure, any branch that goes from a primary basic block to a fluff basic block, or vice-versa, is redirected to a long branch stub. The target of the long branch stub will be the original target basic block. A long branch stub is used because the fluff procedure will most likely be located too far from its primary counterpart for a short branch to reach. Since the paths between primary and fluff code were never

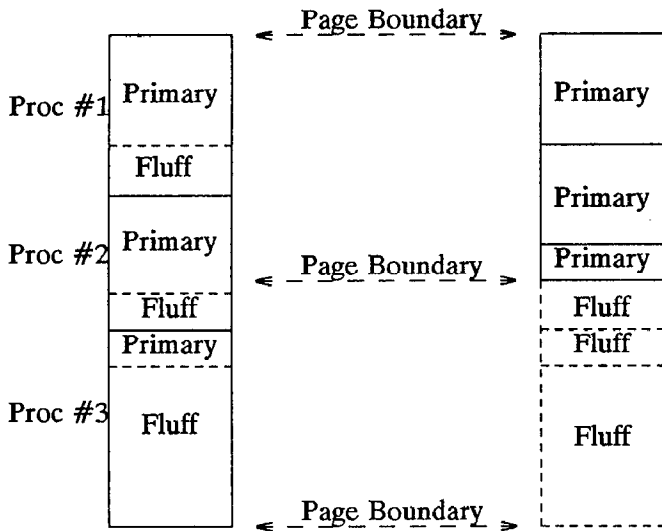


Figure 5: Procedure Splitting Example

taken according to the profiling data, the long branch stubs should be infrequently executed and only for non-typical inputs.

The transfer stubs are placed at the end of the procedure that is doing the interprocedural branch. Placing the stub at the end of the procedure instead of converting the original branches into long branches in-line was chosen for simplicity. If there are multiple branch sites that have the same branch target, a single long branch stub is shared to save space.

An example of an *if-then-else* sequence where the *then* clause is considered to be fluff is shown in Figure 6. The left half shows the traditional ordering of basic blocks while the right shows the ordering after Procedure Splitting has been applied.

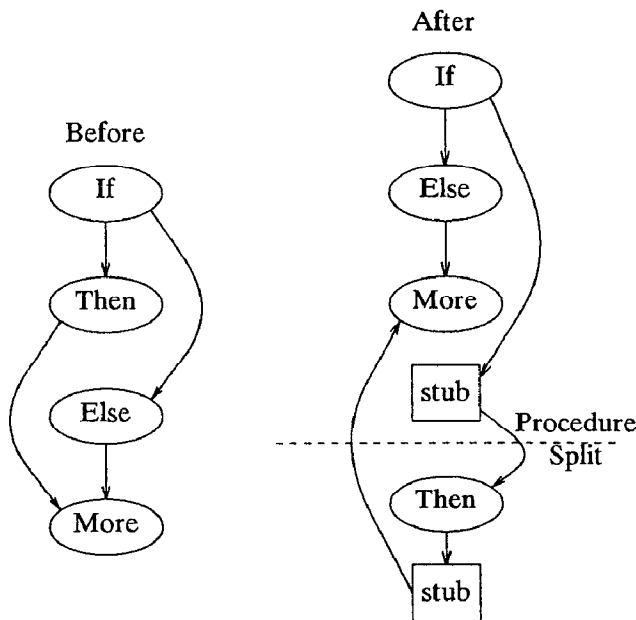


Figure 6: Insertion of Stubs

6. Results

We have implemented the algorithms presented in this paper as prototypes in the PA-RISC optimizer and linker and have successfully restructured several applications written in C, Fortran and Pascal. Because most of the data presented is based on early measurements, at times it may appear to be lacking in a fine-grain analysis. The primary goal of this research was to determine whether the code positioning can deliver a noticeable improvement to the end user. As this project evolves, a more detailed examination of what is happening at the platform level is planned.

Table 1 summarizes some important characteristics of the HP-UX Series 800 platforms used for benchmarking. For configurations that contain a split cache, the numbers show the sizes for the instruction cache followed by the sizes for the data cache.

	Platform		
	825	835	840
VMIPS	9.8	14.8	8.7
Main memory	8Mb	40Mb	24Mb
Cache:			
Unified or Split	Unified	Unified	Split
Size	16Kb	128Kb	64Kb/64Kb
Associativity	1	2	1/1
Line Size (bytes)	32	32	16/16
Lines per Way	256	2K	4K/4K
Clean Miss (cycles)	27	27	7/7
Dirty Miss (cycles)	27	27	14

Table 1: Hardware Configurations

The benchmarks analyzed are summarized in Table 2 and 3. Because of our implementation model, we are able to restructure both multiple source file and multiple language applications. In addition, we do not have to restructure an application for each configuration measured given that our positioning algorithms are platform independent.

Name	Language	Source Lines	Code Size (bytes)
Othello	Pascal	1133	82130
Lex	C	2794	88064
Yacc	C	2847	90112
Simulator	C	21261	323168
Pascal	Pascal/C	312500	2225814

Table 2: Benchmark Characteristics

Name	Description	Primary Inputs
Othello	Board game	Various games
Lex	Lexical analyzer	Internal scanner generator
Yacc	Parser generator	C-front grammar
Simulator	PA-RISC Simulator	Othello/Pascal compiler
Pascal	Pascal compiler	Compiler sources

Table 3: Benchmark Description and Inputs

In table 4, the percentage improvement in user time over full optimization is shown.†

Benchmark	Platform		
	825	835	840
Othello	10.2	2.1	4.0
Lex	10.7	9.3	11.3
Yacc	2.7	0.8	1.8
Simulator	26.0	5.5	14.9
Pascal		7.6	9.8

Table 4: Percentage Performance Improvement

The restructured version of these applications had all three positioning methods applied: basic block positioning, procedure splitting, and procedure positioning. Although procedure positioning was applied to all procedures, including run-time library routines, basic block positioning was only done for the application source code. These numbers represent optimal throughputs in that the same inputs were used for both the collection of profiling data and final benchmarking. The data shows an improvement in user time ranging up to 26 percent.

Table 5 gives a breakdown as to how specific steps in our prototypes improve program performance. Each column of data gives the percentage improvement in user time over full optimization. The third column (PP) shows the results for procedure positioning only. The results in the fourth column (BBP) show improvements for basic block positioning only. The fifth column (BBP-PS) is basic block positioning along with procedure splitting. Finally, columns six (BBP-PP) and seven (BBP-PS-PP) are equivalent to columns four and five with procedure positioning additionally applied.

Benchmark	H/W	PP	BBP	BBP PS	BBP PP	BBP PS PP
Othello	825	7.2	7.9	8.7	8.7	10.2
	835	0.0	1.6	1.6	2.1	2.1
	840	0.9	4.7	4.4	4.2	4.0
Simulator	825	8.2	16.2	13.9	20.3	26.0
	835	0.0	4.5	5.5	4.5	5.5
	840	9.9	0.6	13.8	13.9	14.9
Pascal	835	1.9	5.6	6.9	6.9	7.6
	840	4.3	1.8	9.2	9.3	9.8

H/W Hardware Platform PP Procedure Positioning
 BBP Basic Block Positioning PS Procedure Splitting

Table 5: Percentage Improvement by Prototype

As expected, the largest performance benefit comes from basic block positioning alone. However, there are anomalies in the collected data. For example, the performance for Pascal and the Simulator running on the 840 was only slightly improved using basic block positioning alone whereas when

applied along with procedure positioning, there was a much greater improvement. It is possible that basic block positioning alone has created some new problem such as a collision in the cache that offsets the expected improvement. When procedure positioning is additionally applied, the collision is then alleviated.

When we first began our research, we did not expect to acquire any measurable performance improvements for applications as small as Othello. We were surprised when, in fact, there was a significant performance benefit due to positioning. Table 6 shows a breakdown of where the performance gains were acquired for Othello. The second column is the percentage improvement in the specific individual measurement. The third column is the percentage of total improvement relative to other improvements. This information was gathered using the PA-RISC Simulator simulating an 825 configuration. For the purpose of illustration, we assumed a 1 cycle branch penalty for conditional branches taken the wrong way.

Parameter	% Improvement	% of Total Improvement
I-cache Misses	98.2	57.8
Branch Penalty Cycles	42.3	24.0
Executed Instructions	2.7	18.2

Table 6: Performance Breakdown for Othello

A nice side effect, but not an original goal of this optimization technology, is the reduction in the number of executed instructions. This is due to two reasons. First, the straightening out of *if-then* sequences reduces the number of nullified delay slots (which count as wasted instructions). Second, moving infrequently executed basic blocks (e.g. an *else* block) out of the mainstream removes the unconditional branches otherwise needed to branch around these infrequent basic blocks. This could mean that basic block positioning alone could achieve 2-3 percent in performance simply due to reducing the number of executed instructions.

6.1. Specific Observations on Procedure Positioning

One of the benefits gained from procedure positioning a large application is a reduction in the number of executed long branches. This section shows how we have reduced the number of executed long branches for procedure positioned Pascal and Fortran compilers.

In addition to an original non-positioned compiler, a number of procedure positioned compilers were produced based on different sets of profiling data:

- *oth*: The profiling data used was from compiling othello *without* optimization.
- *oth.-O*: The profiling data used was from compiling othello *with* full optimization.
- *multiple.-O*: For Pascal, the profiling data used was from compiling null.p, othello.p, talias.p, and flatpm.p *with* full optimization.† For Fortran, the profiling data used was from compiling null.f, othello.f, cae.f, and spice.f *with* full optimization.

† Talias.p is a compiler front-end routine, flatpm.p is a Process Management routine from MPE/XL.

† Most timing measurements were done using the UNIX‡ *time* command.

‡ UNIX is a trademark of AT&T Bell Laboratories.

Table 7 shows the number of static long branches necessary to link the compilers based on the collected profiling data.

Positioned Compiler	Pascal	Fortran
Original	13383	2635
oth	19347	12447
oth.-O	20896	13570
multiple.-O	22211	12986

Table 7: Static long branch stubs

The number of static branches required has increased by up to 65 percent for Pascal and up to 400 percent for Fortran!

Tables 8 and 9 show the effects on the number of executed long branches.

Positioned Pascal Compiler	Input and Options				
	othello.p no opts	othello.p -O	null.p -O	talias.p -O	flatpm.p -O
Original	101126	152092	824	2104156	6920411
oth	1884	127534	241	1773933	4424816
oth.-O	26489	10443	740	1828036	3639081
multiple.-O	34251	22802	500	139149	434156

Table 8: Pascal dynamic long branches

Positioned Fortran Compiler	Input and Options				
	othello.f no opts	othello.f -O	null.f -O	cae.f -O	spice.f -O
Original	35611	37175	310	349383	1586339
oth	2618	60574	505	795580	3256855
oth.-O	18753	6558	136	350337	957727
multiple.-O	26759	15388	181	69830	226851

Table 9: Fortran dynamic long branches

The **BOLD** numbers represent *optimal* runs in that the input files used for final analysis were the same as those used for profiling. For these cases, the decrease in the number of executed long branches varies from 85 to 98 percent for Pascal (excluding compiling null.p) and from 80 to 92 percent for Fortran as compared to the non-positioned compilers.

There are two additional pieces of information to gather from the data above. First, although there was an increase in the number of static long branch stubs, the number of executed long branch stubs was significantly reduced. This means that efforts directed only towards reducing the static count may give unexpected results. Second, for Pascal there was a general decrease in the number of executed long branches using any input on any of the positioned compilers. While using the *oth* version of Pascal, which was positioned using profile data for compiling othello.p *without* optimization, there was a decrease varying from 16 to 75 percent while compiling other source files *with* optimization. In other words, it appears that we have not worsened the performance (in regard to procedure locality) for non-typical inputs.†

This was not true for all cases in Fortran. However, the non-positioned Fortran compiler already had some hand optimiza-

tions on the ordering of relocatable files to produce better procedure locality.

All procedure positioned Pascal and Fortran compilers had a general run-time improvement ranging from 1 to 6 percent even for non-profiled inputs. For Fortran, this was true even though the number of executed long branches went up in some cases. For both Pascal and Fortran, the reduction in the number of executed long branches did not account for the total throughput improvement. These facts point to something else contributing to a gain in performance.

Although data is not presented, it appears that paging, TLB, or most likely, cache performance, have been improved by our "closest is best" procedure positioning strategy.

6.2. Specific Observations on Basic Block Positioning

The primary goal for the basic block positioning algorithm is to produce straight-line sequences of code that make more effective use of the cache. A secondary goal was to minimize the number of taken penalty branches.

Using simulated data for measuring the performance of Othello running on an 825 configuration, the following results were observed:

- The number of executed penalty branches was reduced by 42 percent.
- The average number of instructions executed before taking a branch increased from 6.19 to 8.09, a 31 percent improvement.

By executing more instructions before taking a branch, there is a better chance that we are making more effective use of the cache.

6.3. Specific Observations on Procedure Splitting

The goal of procedure splitting is to minimize the size of the primary procedure by separating the unused code from the primary code. As previously discussed, some code must be added to both the primary and fluff procedures to properly handle transfer of control between the two procedures.

The second and third columns in Table 10 give a breakdown on the number of instructions that were considered to be primary and fluff code, respectively, based on the profiling data used. The fourth column is the number of long branch instructions that had to be added to the primary procedures to properly handle a transfer of control. The last column is the ratio of the number of long branch instructions added to the primary procedure versus the number of fluff instructions moved. The ratio shows that a large number of instructions have been moved with a very small overhead in the number of instructions added.

Benchmark	Primary Instrs.	Fluff Instrs.	STUB Instrs.	Ratio Fluff : STUB
Othello	16072	582	42	13.8 : 1
Simulator	31129	47494	1382	34.4 : 1
Pascal	189422	330312	11794	28.0 : 1

Table 10: Fluff Statistics

6.4. Compile-time Costs

Adding profiling code during compilation is expected to increase the time necessary for compilation. In addition, the added complexity of basic block positioning should also result in an increased compilation cost. Although it was not a primary concern of our prototype effort, data is presented for compilation costs.

The compilation costs required to do basic block positioning are shown in table 11.‡ The second column is the time required to compile the application with optimization. The third column is the time required to add profiling code. Note that we still perform full optimization when adding profile support. The last column is the time required to perform basic block positioning along with optimization.

File	Full Opt	Adding Profiling	Basic Block Positioning
othello.p	22.7	37.0	24.4
talias.p	192.8	356.7	193.4
lex.c	124.8	409.9	148.1
yacc.c	103.6	243.2	121.9

Table 11: Compilation Costs (seconds)

As can be seen, the additional time required to perform basic block positioning compared to full optimization alone is fairly small. However, there is a 2 to 3 factor slowdown in the time required to add profiling support. Compilation time is strongly related to the number of basic blocks and adding measurement code more than doubles the number of blocks for a procedure. There are methods that may be employed to minimize the number of counters needed to obtain accurate profiling data [Sar89]. These should help reduce the compilation costs by reducing the number of additional basic blocks needed.

6.5. Run-time Costs

In addition to compilation costs, there is an expected slowdown in the throughput of an application that has had profiling code added to it. Again, this was not a primary concern of our development, but data is presented.

Using a control input, the second column of table 12 is the time required to run an application when it was compiled with full optimization only. The third column is the time required to run the application when it was compiled and linked with profiling code.

Benchmark	Full Optimization	Profiling Code Added
othello	9.9	24.7
lex	4.2	19.6
yacc	6.1	18.5
Pascal	154.7	321.6

Table 12: Run-time Costs (seconds)

‡ The compilers used to collect this timing information were our prototype compilers and were not optimized.

As with compilation costs, there is a significant overhead in the run-time performance of an application that has been instrumented with profiling support. The methods described to reduce compilation costs would also help reduce these run-time penalties.

7. Positioning Expectations

From our experiences, procedure positioning gives the most improvement when the original application has a large number of procedures with poor locality in its call graph. Applications that have already been structured to improve locality show far less improvement.

Basic block positioning and procedure splitting give the most improvement when there are lots of conditional branches around relatively unused portions of code. When examining the code for the Simulator to determine why there was such a large performance improvement in the restructured version, we noticed that there are a large number of non-looping routines that once entered, perform some simple operation, branch to near the end of the routine, perform some other operation, and then return. This type of behavior puts excessive pressure on the cache system and is common in many real applications.

Because our methods are currently used for positioning code only, applications that spend much of their time in data intensive loops do not show as much improvement. A large amount of data cache misses will shadow any benefits gained from code positioning. In particular, when we positioned the optimizer components of the compiler, we did not see much of an improvement because it spends much of its time traversing large data structures within tight loops.

8. Issues

This section describes issues that were not addressed because these were not necessary in order to implement the technology in a prototype. However, these should be addressed for any product effort.

- *Two-pass Compilation:* For basic block positioning, the current prototype requires a compilation pass for measurement and a separate pass for positioning after profiling data has been collected. McFarling [McF89] discusses using profile information along with program structure obtained directly from standard object files to create positioned object files.

Conceivably, both basic block profiling and positioning could be done in the linker or a separate tool altogether. This would remove the need for multiple compiles, but an extensive amount of knowledge on program structure would have to be built into the tool.

- *Debugging Positioned Code:* Most high level symbolic debuggers have difficulties dealing with optimized code. When programmers started using full optimization, they could no longer use the symbolic debuggers to debug their code. Over time, users learned how to find markers, such as procedure calls, in the assembly code while using low level debuggers such as *adb* or *nmdebug*†. Using these markers, users are able to find the code of interest in order to solve the problem at hand.

† *Adb* is a low-level debugger on UNIX while *Nmdebug* is a low-level debugger on MPE/XL.

With basic block positioning, these markers may no longer be useful. For example, if the code of interest follows a procedure call in the source, users often look for the procedure call in the assembly language to find the desired code. In the positioned code, the procedure call marker may now be located almost anywhere in the procedure with no relation to the point of interest. In fact, if procedure splitting has been done, they may not be in the same procedure at all.

The general problem of debugging optimized code has been addressed elsewhere [CMR88] and we feel that the same techniques could be used with positioned code to hide the underlying complexity from the end user.

- *Reuse of Data:* This is an important issue because users are not going to be willing to recollect profiling information and position after making minor changes. For procedure positioning, the current scheme of producing an ASCII list for the ordering of the procedures may be adequate for small changes. If a user adds or deletes a small number of routines (or if the dynamic call graph changes slightly) the previously generated procedure order will still be useful, but maybe not the most optimal.

For basic block positioning, the problem is much more complex when a user alters the structure of a procedure. If the profiling data no longer maps to the basic block structure, it would be very difficult to decide how to position the new structure. At a minimum, these altered procedures could be skipped over during positioning with a warning issued, but procedures that have not been modified could still be handled.

- *Representative Inputs:* We did not address the problem of using representative inputs to collect profiling data. There is some concern that program performance could be degraded for typical inputs if the wrong set of inputs is used to collect profiling data. From our analysis, it appears that many programs have a central trunk of execution that is executed regardless of the inputs supplied. By positioning this trunk, a general improvement has been measured regardless of the inputs used.

9. Additional Opportunities

The following is a list of additional opportunities that may lead to an increase in application performance or other benefits. This list is by no means considered to be complete:

- *Alternative Positioning Algorithms:* Alternative positioning algorithms for determining the ordering of procedures and basic blocks.
- *Modeling the Cache:* We did not take any machine dependent parameters into account when doing positioning. For procedure positioning, if we knew the total cache size for the target machine, we might use an algorithm that tries to minimize cache overlap instead of "closest is best".

For basic block positioning, if we knew the cache line size, we might try to cache align branch targets.

- *Conditional Branch percentage:* Hwu and Chang [HC89] did some analysis on when the less dominant block of an if-then-else structure should be moved out of line. For example, you may not want to move any block if the less dominant block is entered 45 percent of the time relative to the dominant block, but you probably do if it is entered only 30 percent.

In our prototype, the less dominant, no matter what the difference, was moved out of line, since that was always better in terms of simple cycle counts, not including cache misses.

- *Fluff Block Size:* Any basic block that was marked as a fluff basic block was moved into the fluff procedure, regardless of the size of the basic block. Given that a long branch stub consists of two instructions anyway, it may not be beneficial to move very small blocks.
- *Inlining Interprocedural Branches:* When doing procedure splitting, instead of redirecting the simple branches to long branch stubs for interprocedural transfer, these branches could be turned into long branch sequences in-line. This would be a static and dynamic saving.
- *Interprocedural Branch Distance:* For smaller applications, there may not be a need for long branches to branch from primary to fluff regions. The original branch could be left unmodified, even if its target was in another region, if the branch distance was known to reach.
- *User Directives:* It may be beneficial to allow user directives in the code specifying heavily executed or infrequently executed basic blocks or procedures. This information could be used to do positioning without the need of profiling data. Of course, this depends on the user having accurate knowledge of his program.

10. Conclusion

In our work on code positioning, we developed algorithms and prototypes to position code based on execution profile guidance. The optimization techniques that we implemented in the prototypes vary widely in their effectiveness based on the application and the particular PA-RISC implementation. Performance improvements ranging from 2 to 26 percent over current optimizations have been measured with an average improvement of 8 to 10 percent. More of this gain is due to positioning code at the basic block level rather than at the procedure level.

11. References

[CMR88]

Coutant, Meloy and Ruscetta, "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code," *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, Vol. 23, No. 7, July 1988, pp. 125-134.

[DV87]

Davidson and Vaughan, "The Effect of Instruction Set Complexity on Program Size and Memory Performance," *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987, pp. 60-63.

[Fer74]

Ferrari, "Improving Locality by Critical Working Sets," *CACM*, Vol. 17, No. 11, November 1974, pp. 614-620.

[Fer76]

Ferrari, "The Improvement of Program Behavior," *Computer*, Vol. 9, No. 11, November 1976, pp. 39-47.

[GKM82]

Graham, Kessler and McKusick, "gprof: a Call Graph Execution Profiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No. 6, June 1982, pp. 120-126.

[Har88]

Hartley, "Compile-Time Program Restructuring in Multiprogrammed Virtual Memory Systems," *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, November, 1988, pp. 1640-1644.

[HG71]

Hatfield and Gerald, "Program Restructuring for Virtual Memory," *IBM Systems Journal*, Vol. 10, No. 3, 1971, pp. 168-192.

[HC89]

Hwu and Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proc. 16th Sym. on Computer Architecture*, Jerusalem, Israel, May 1989, pp. 242-250.

[McF89]

McFarling, "Program Optimization for Instruction Caches," *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 183-191.

[Sar89]

Sarkar, "Determining Average Program Execution Times and their Variance," *Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, Vol. 24, No. 7, July 1989, pp. 298-312.