Trace-directed program restructuring for AIX executables

by R. R. Heisch

This paper presents the design and implementation of trace-directed program restructuring (TDPR) for AIX® executable programs. TDPR is the process of reordering the instructions in an executable program, using an actual execution profile (or instruction address trace) for a selected workload, to improve utilization of the existing hardware architecture. Generally, the application of TDPR results in faster programs, programs that use less real memory, or both. Previous similar work [1–6] regarding profile-guided or feedback-directed program optimization has demonstrated significant improvements for various architectures. TDPR applies these concepts to AIX executable programs at a global level (i.e., independent of procedure or other structural boundaries) running on the POWER, POWER2™, and PowerPC 601™ machines and adds the methodology to preserve correctness and debuggability for reordered executables. Using the prototype tools developed for this effort on a selection of both user-level application programs and operating system (kernel) code, improvements in execution time of up to 73% and reduced instruction memory requirements of up to 61% were measured. The techniques used to restructure AIX executables are discussed, and the performance improvements and memory reductions measured for several application programs are presented.

Introduction

Today's high-performance computer memory architectures are optimized for programs which exhibit high spatial and/or temporal locality for both instructions and data. Memory hierarchies have evolved in an attempt to minimize cost and maximize performance by exploiting this "locality of reference" program characteristic. Similarly, design assumptions are typically made regarding other program characteristics (such as branching behavior) which result in processor designs optimized for those assumed characteristics (such as branch prediction).

As long as these program assumptions hold, processor performance is maximized. However, when a program deviates from these assumed characteristics, the processor architecture is inefficiently utilized, which usually leads to reduced performance or excessive use of real memory.

While hardware design tradeoffs are made on the basis of software-related assumptions, compilers attempt to

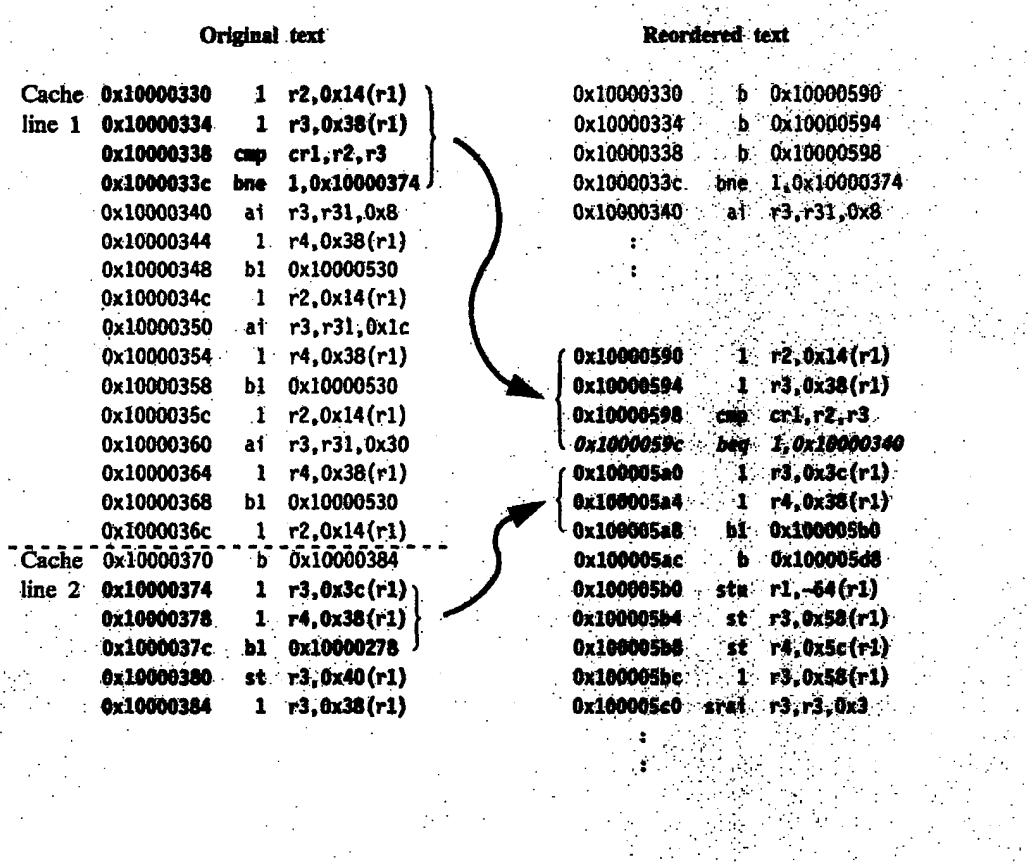

Figure 1

Reordering example.

generate "optimum" code targeted for a specific hardware architecture (including the memory hierarchy) on the basis of similar program assumptions. However, compiler optimizations are usually limited to a purely static analysis of a program which includes speculation as to how a program will probably execute on a given hardware platform. Additionally, since many programs result from binding together multiple, separately compiled (or assembled) object modules, the compiler does not usually have a "global view" of the final organization of the executable image and therefore cannot perform a truly global optimization.

TDPR effectively "closes the loop" in the optimization process. It attempts to further optimize a program by collecting information on the actual behavior of a program while it is executed and using that information to reorder and modify instructions across the entire executable program image to optimize the use of the hardware.

Consider the following simple example of poor program locality for a typical high-level language code sequence:

```
if (x == y)
        {
        /* Error handler code */
        }
/* Otherwise, execution continues here */
```

In this code sequence, the error path (taken when variable x is equal to variable y) is usually not executed (information which is not known at compile time). **Figure 1** shows the resulting assembler code generated for a typical code sequence of this type. The example represents a machine with 16 instructions per instruction cache line.

Notice that although only the first four instructions are usually executed [the instructions for the if (x == y) statement], the remaining unexecuted instructions (representing the error handler code) are also loaded into the cache. Since the minimum allocatable unit of a cache (typically a cache line) is usually much larger than a single instruction, poor program locality results in higher miss rates, and therefore reduced performance, due to inefficient cache utilization. Similarly, real memory space may be wasted on instructions which are usually not executed but, due to their proximity to frequently executed code, are loaded when a real page is allocated.

Figure 1 also shows the results of reordering the instructions according to the way in which they are executed. On the basis of information collected at run time, the frequently executed code paths are grouped together. The result is improved performance [due to reduced instruction cache and TLB (translation lookaside buffer) miss rates] and a reduction in run-time memory requirements (due to improved utilization of real memory pages).

Also, the conditional branch instruction has been recoded with a different branch target address and the opposite (reversed sense) condition code (from a *BNE Target_Address* to *BEQ Fall_Thru_Address*). This illustrates an additional opportunity to improve performance, on the basis of actual program behavior, by reducing inefficient use of available hardware optimizations (which, in this case, are reduced pipeline stalls due to incorrectly predicted-not-taken branches).

Another improvement, which results indirectly from stringing together frequently executed code paths, is that of reduced collisions in an N-way set-associative cache. If more than N instructions in a highly executed code loop map to the same cache congruence class, constant cache misses will occur because of the thrashing which results from these collisions. Reordering the instructions in a program according to the actual execution path potentially produces additional performance improvements by reducing "conflict misses" in an N-way set-associative cache.

TDPR process overview

The process of applying trace-directed program restructuring is illustrated in **Figure 2**. First, the executable program to be restructured is run for the desired workload (W) while an instruction address trace (or execution profile) is captured and analyzed. The result of this analysis is an address reorder list which represents the "optimal" ordering of the instructions in that executable program image for the given workload. Second, the address reorder list and the executable program file are used to create a new, restructured, executable by

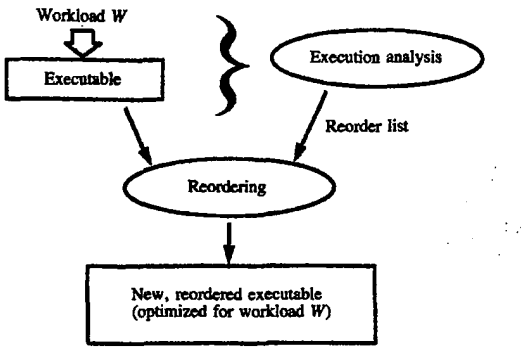

Figure 2

TDPR process diagram.

reordering the instructions from the original program in the sequence specified in the reorder list.

The reordered executable resulting from applying the TDPR process will exhibit varying degrees of performance improvement and/or reduced instruction memory requirements when run on workload W (or similar workloads).

Design and implementation of TDPR for AIX executables

The design and implementation of trace-directed program restructuring for AIX® executable programs entails solving two major problems: 1) managing dynamically calculated branches (computed goto's) and 2) generating an "optimal" address reorder list. Once these problems are solved, the remainder of the effort revolves around the fairly simple repositioning and accounting required to build the reordered executable.

In this implementation, the minimum reorderable unit is the basic block (a basic block is defined as a sequence of instructions that has exactly one entry point and exactly one exit point). The addresses specified for TDPR are the addresses of the first instruction in the basic block. When a basic block is moved while reordering an executable, all of the instructions in the basic block are moved together.

- *Managing dynamically calculated branches*

The branch target or destination address of a dynamically calculated branch (DCB) is calculated as a program runs and is usually difficult if not impossible to determine statically. For the POWER, POWER2™ and PowerPC 601™

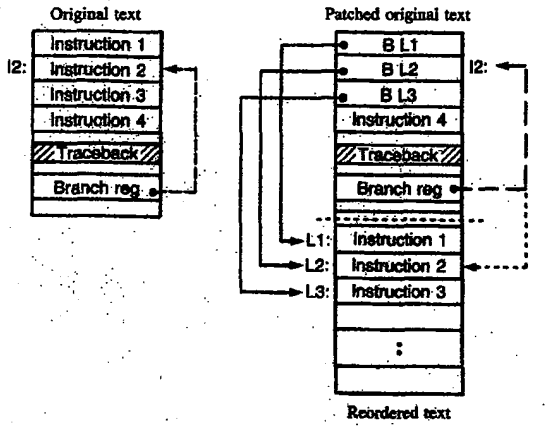

Figure 3

Managing dynamically calculated branches.

processors, the DCB takes the form of a branch-to-register instruction. In order to move instructions during TDPR, some mechanism must be provided to eliminate the problem of a DCB calculating and branching to the address of an instruction that has been moved. One such mechanism would be to attempt to recognize all possible types of DCBs generated for some subset of all compilers (and compiler versions) used to create the executable programs. The problem with this approach is that it is not fail-safe, and program functionality or correctness cannot be guaranteed because of the possibility of unanticipated code sequences (such as might arise with different compiler versions or with user-written, "nonstandard" assembler programs).

The mechanism developed to manage dynamically calculated branches for this implementation of TDPR is illustrated in **Figure 3**. The idea is to keep the original text (instruction) section intact except for instructions that are reordered (i.e., moved during TDPR). Reordered instructions are appended to the end of the original executable (in the "reordered text area") and are replaced (in the "original text area") with branches to the new addresses where the instructions have been moved.

For example, Instructions 1, 2, and 3 in the original text section shown in Figure 3 have been moved to locations L1, L2, and L3 in the reordered text area, and the original instructions in the patched original text area have been replaced with branches (B) to locations L1, L2, and L3, respectively. Instruction 4 and the Branch reg (branch to register) instruction, which are not part of a frequently executed code path in this example, are not moved. Additionally, all traceback entries (which are embedded at the end of each procedure for program debug) are removed from highly executed code paths (i.e., not moved with reordered code) but are maintained in the original text section for debuggability. With this mechanism in place, if an unanticipated DCB attempts to branch to the address of a moved instruction (such as the Branch reg to location I2), it will simply the encounter the branch (B L2) to the new location of the instruction and then branch to that new location, thus preserving functionality.

Although this technique for managing DCBs does maintain functionality for most programs (high-level language and assembler alike), it can be undesirable from the perspective of performance and memory utilization because of the double branch sequence, resulting from undetected DCBs, which usually forces two memory pages to be touched. However, the vast majority of DCBs found in AIX executables are due to 1) the C switch/case statement (which typically generates a branch table in the program constant area) and 2) calling a function through a pointer (which uses a function descriptor in the program data area). This double branch sequence can usually be eliminated by updating the addresses of moved instructions in the branch tables and function descriptors with the new reordered addresses. In this implementation of TDPR, both the branch tables and the function descriptors are scanned for the addresses of moved instructions and are (optionally) modified with the correct reordered addresses.

Using this mechanism for managing DCBs, a branch to the reordered text area is executed once when a program first begins; from that point on, execution is constrained to the optimized reordered text area. If, however, an unanticipated DCB (i.e., one that is undetectable and/or cannot be modified) is encountered during program execution, the performance improvement gained by reordering may degrade slightly, but the program will continue to produce the expected results.

- *Generating an address reorder list for TDPR*

To apply TDPR to a program, the instruction address trace (or profile) collected during program execution must first be analyzed to determine an "optimal" basic block ordering which will result in the maximum speedup (execution time improvement) and/or memory requirement reduction. Determining the optimal ordering of the basic blocks in a program is a challenging problem. The approach used here (similar to that discussed in [3]) is to attempt to identify the most frequently executed paths through the code by building a directed flow graph (DFG) from the address trace (or profile) collected during program execution.

The DFG consists of a node for every basic block with an associated count of the number of times that basic

block was executed. Additionally, each node has one or more edges (or pointers), with associated counts, to the node of the basic block or blocks which are executed next. For example, **Figure 4** shows the DFG generated for the following sequential instruction address trace:

200, 800, 100, 800, 400, 200, 800, 400, 200, 800, 400, 200, 800, 400, 200, 800, 400, 200, 800, 400, 200, 600, 200, 600, 200, 700

In this simple example trace, the basic block at address 200 is executed first, followed by the basic blocks at addresses 800, 100, 800 and so on up to the last basic block at address 700. The basic block at address 200 was executed a total of nine times, six of which ended in transferring control to the basic block at address 800, two going to 600, and one to address 700. As can be seen in the DFG, the frequently executed or "hot" code path for this address trace is the sequence 200–800–400.

The algorithm used in this implementation for generating the reorder list is described as follows:

1. Build the DFG from the instruction address trace or profile as shown in Figure 4.
2. Provide the following alternate methods for traversing the DFG to produce the address reorder list:
   a. Starting with the most frequently executed basic block, follow the most frequently executed paths until a cycle is detected (i.e., a previously visited basic block). As each basic block node is visited in the DFG, append the basic block address to the address reorder list. When a cycle is detected, restart the process at the next most frequently executed address. This is the $np = 0$ option.
   b. Same as (a), except that when a cycle is detected, back up one node and then go visit each next most frequently executed basic block. This is the $np = 1$ option.
   c. Same as (b) except that when backing up to visit each next most frequently executed basic block, visit only those nodes which are executed next more than N times. This is the $np = N$ option.

Table 1 shows the address reorder lists generated for the DFG shown in Figure 4 using this algorithm.

While the slight differences in the reorder lists shown may appear inconsequential, the performance differences can be significant for large code sequences which approach or exceed the size of the instruction cache. Selecting the appropriate *np* option, however, is usually a matter of trial and error (although the $np = 0$ option usually provides the best speedup for most programs in this implementation).

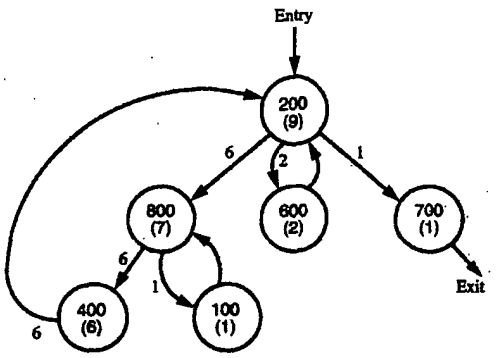

Figure 4

Directed flow graph example.

Table 1 Reorder lists for different *np* options.

np = 0	*np = 1*	*np = N = 2*
200	200	200
800	800	800
400	400	400
600	100	600
100	600	
700	700	

• *Maintaining basic block movement*

The remainder of the implementation involves the housekeeping required to accommodate the movement of basic blocks within the program while maintaining the expected functionality. In this implementation of TDPR, basic blocks are moved sequentially in a single pass, as specified in the address reorder list.

The diagram shown in **Figure 5** illustrates the movement of a basic block (BBn) from its original position in the program to its new location (in the reordered text area). In this example, basic block BBn branches to the basic block at address L1, and two basic blocks (B1 and B2) both branch to BBn. When a basic block is moved, both the branch out of the basic block (if it exists) and all branches into the basic block must be adjusted.

Basic block movement is managed by maintaining a dual entry log for each basic block in the original text section. The first entry is an address that indicates where the basic block for this log entry has been moved. The second entry is a pointer to a list of all basic blocks that branch to the basic block for this log entry. Whenever a basic block is moved, the moved_to log entry for that block is assigned

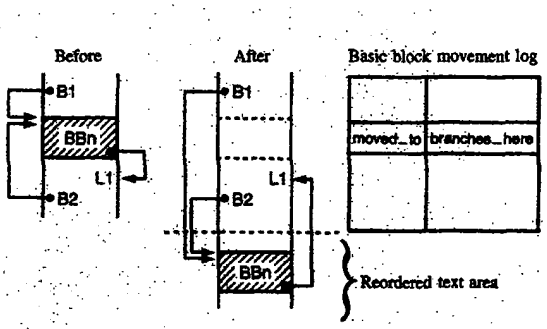

Figure 5

Tracking basic block movement.

Table 2 Measured program speedups (%).

Program	*POWER*		*POWER2*	*PowerPC 601*	*{y-bit}*
	(8KIC)	(32KIC)			
ksh	+45	+14	+13	+20	{+4}
awk	+19	+9	+10	+11	{+2}
vi	+13	+8	+6	+22	{+2}
sed	+7	+5	+4	+6	{+3}
SPEC 022.li	+20	+5	+4	+9	{+2}
SPEC 072.sc	+11	+4	+1	+5	{+2}
SPEC 056.ear	+9	+9	+4	+2	{−1}
RDBMS1 TPC-A	—	—	+15	—	
RDBMS1 TPC-B	+17	+19	—	—	
RDBMS2 TPC-C	—	+12	—	—	

the new address of the basic block, and all basic blocks which branch to the block to be moved (indicated by the branches_here entry) are adjusted to branch to the new location.

• *Branch replacement*

During the course of moving basic blocks while applying TDPR, the opportunity or requirement may arise to modify the branch that usually terminates a basic block. This modification may come in one of the following forms:

1. Changing the sense of a conditional branch (and modifying the branch target address) to improve hardware branch prediction.
2. Converting to a branch sequence to handle "branch target out of range" and "moved fall-through code" problems. The "branch target out of range" problem occurs if a target address is not reachable from the address of a branch instruction (because of the size of the branch displacement field in the instruction); "moved fall-through code" problems occur if the code which follows a basic block is moved elsewhere.
3. Adjusting branch target addresses due to moved basic blocks.
4. Eliminating a branch instruction altogether.

The branch replacement algorithm in this implementation consists of two main cases: 1) branch-to-register, and 2) branch immediate (not to register). For the branch-to-register case, if the basic block at the fall-through address (i.e., immediately following the basic block) will not be moved next, an additional branch to the fall-through code is inserted (if needed). For the conditional branch immediate, depending upon whether the basic block at the fall-through or target address is the next basic block in sequence, the branch condition is adjusted (if possible and necessary) such that the branch will be predicted correctly most often (where the sequence of the basic blocks from the reorder list implies the most frequently executed path). Also, the branch target range for existing or modified branches is examined, and unconditional "far" branches are added if the branch target or fall-through address is out of range.

TDPR for user-level programs

Applying TDPR to user-level application programs involves the following:

1. Reading/decoding the AIX XCOFF (eXtended Common Object File Format) executable program image and collecting the different sections (data, text, etc.).
2. Reordering the text section. This is done by applying the techniques described above and appending the reordered code to the end of the original text section. The size of the text section specified in the XCOFF text header is adjusted accordingly.
3. Applying any "fix-ups" to the branch tables (for switch/case statements), to the function descriptors in the data section (for function calls through pointers), and to any other XCOFF sections (such as debug information).
4. Writing out the new executable XCOFF file image of the reordered program.

TDPR results

The results measured for reordering user-level applications are shown in the tables which follow. The RISC System/6000® Model 530 was used for POWER 8KB instruction cache (8KIC) measurements, and the RISC System/6000 Model 570 was used for POWER 32KIC measurements.

Table 2 shows the speedups measured for the 8KIC and 32KIC POWER machines and for the POWER2 and 601 machines. All speedups shown were calculated by

comparing the execution time of the original program to that of the reordered program for the same workload on the same machine. Two different, commercially available, relational database management systems (RDBMS1 and RDBMS2) were used for the TPC-A,™ TPC-B,™ and TPC-C™ tests.

It is important to note that each of the programs shown in Table 2 was reordered and tested on the exact same workload specific to that test; results for cross-workload measurements are presented in Table 5, shown later.

Also shown in Table 2 are the results of adjusting the 601 branch predict bit (y-bit) using the actual execution profile data collected for these programs. The 601 processor provides a bit in the conditional branch option field that allows software to adjust the branch prediction algorithm used for conditional branches. TDPR was not applied for these y-bit tests. Actual branch-taken/not-taken percentages were calculated from the execution profile data, and the y-bit was adjusted accordingly to improve the success of hardware branch prediction. These data, along with the hardware monitor results shown below, provide an indication of the amount of speedup due only to improved branch prediction. The 1% performance decrease seen for the SPEC™ 056.ear benchmark is apparently due to second-order cache and branch prediction effects.

The factors contributing to the 17% speedup measured for the RDBMS1 TPC-B test (on the POWER machine) are shown in **Table 3**. These measurements were taken using a POWER hardware performance monitor which provides exact counts for clock cycles, instructions executed, cache and TLB misses, etc., throughout the execution of the program. These data indicate that the application of TDPR provides an improvement in CPI (cycles per instruction) resulting from reduced instruction cache (IC) and TLB miss rates, and reductions in the percentage of conditionally issued instructions (i.e., conditional branches) that were canceled (i.e., predicted incorrectly).

The reductions in text real memory requirements for several user-level application programs are shown in **Table 4**. The changes in memory requirements were calculated using two different methods (shown as *xx*/*yy* in the table). The first number (*xx*) represents the change in the total number of pages required for the execution of the program; the second number (*yy*) indicates the change in the maximum simultaneous pages required during execution. The increases shown for awk and vi are due to missed branch table modifications (as described above), which result in additional text memory pages touched during execution. However, the 61% reduction for the RDBMS1 TPC-B test represents an instruction memory savings for this program of more than 512 KB.

Applying TDPR using the methodology described herein does have the disadvantage of increasing the size of the executable program file (because the reordered text is appended to the original executable). However, in environments where disk space is not an extremely critical resource, trading additional disk storage requirements for both improved performance and reduced real memory requirements is usually desirable. The increases in executable file sizes are also shown in Table 4.

Table 3 Factors contributing to RDBMS1 TPC-B speedup.

Parameter	*Reordered*	*Original*
CPI	2.52	2.98
IC miss	4.20%	5.90%
ITLB miss	0.150%	0.390%
Can/cond	23.0%	52.0%

Table 4 Text working set and executable size changes (%).

Program	*Text memory requirements*	*Executable size*
ksh	−51/−63	+16
awk	−9/+40	+16
vi	+9/−64	+31
sed	−25/−25	+41
SPEC 022.li	−31/−59	+11
SPEC 072.sc	−28/−24	+19
SPEC 056.ear	−18/−48	+8
RDBMS1 TPC-B	−43/−61	+5

Cross-workload effects

One potential problem with applying TDPR is that of determining an appropriate workload to use while reordering a program. If two different workloads exercise a program in a completely different manner, finding a single address reorder list that is optimal for both workloads is improbable. For example, a program is reordered for workload A, and the reordered version is then run on workload A and results in a speedup of Sa. Similarly, a version reordered for workload B is run on workload B and results in a speedup of Sb. Reordering a third version of the program for both workloads A and B together, where the workloads use and exercise the program very differently, and running that version separately on both workloads usually results in speedups of less than Sa and Sb. Also, running a reordered program on workload C, where workload C was not in the set of workloads used to reorder the program, typically also yields little (or possibly negative) improvement if workload C is very different from the other workloads.

For example, **Table 5** shows the cross-workload results for reordering both the awk and ksh executable programs. The three reordered programs for awk are awk.heap (reordered for a heapsort workload), awk.pts [reordered for an awk PTS (performance test suite) workload], and

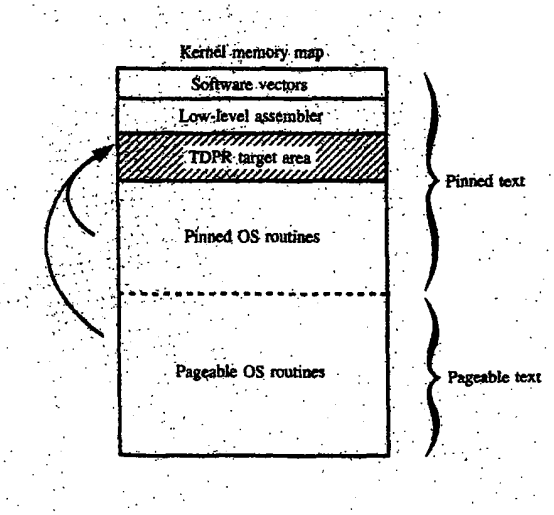

Figure 6

Applying TDPR to the AIX kernel.

Table 5 Cross-workload results.

Workload	*Reordered program speedups (%)*		
	awk.heap	*awk.pts*	*awk.comb*
heapsort	+19	−9	+18
PTS	+18	+22	+18
	ksh.scr	*ksh.sum*	*ksh.comb*
scr1	+21	+3	+18
sum.ksh	+11	+45	+30

awk.comb (reordered for both the heapsort and PTS workloads). The reordered programs for ksh are ksh.scr (reordered for the ksh "built-in" commands workload scr1), ksh.sum (reordered for the sequential summation workload sum.ksh), and ksh.comb (reordered for both scr1 and sum.ksh workloads).

As the data of Table 5 indicate, running the awk.pts program on the heapsort workload (i.e., a workload not used to reorder the program) actually results in a decrease in performance. However, running awk.heap on the PTS workload (again, a workload not used to reorder the program) results in an 18% speedup (slightly less than awk.heap run on the heapsort workload). The combined reordered awk (awk.comb) produces significant speedups for both workloads (although awk.comb running PTS yields less improvement than awk.pts running PTS). The ksh cross-workload results are quite similar to the awk results, with only +3% speedup shown for ksh.sum running the scr1 workload and still significant speedups for ksh.comb on both workloads. However, to achieve the maximum performance improvements (at least for ksh and awk and these simple workloads), the program must be reordered for the exact workload for which it is to be used.

A potential solution to the "cross-workload effect" problem for widely varying workloads is to produce different versions of the program which are each optimized for specific workload types. Then, knowing what workload type is to be run, the reordered version of the program that is optimized for that workload type is used.

TDPR for kernel/kernel extensions and device drivers

In addition to user-level executable programs, significant improvements can also be achieved by applying TDPR to AIX base operating system (kernel) code, kernel extensions, and device drivers with the following special considerations. Implementing TDPR on executable images is not well suited to programs which utilize self-modifying or otherwise position-dependent code because of the difficulty in detecting and correcting for modifications to code that has been moved. A form of position-dependent code can be found in system-level software (such as the base kernel, kernel extensions, and device drivers of AIX) which utilizes pinned instruction memory. Pinned memory is memory (in a virtual memory system) that is never "paged out" (i.e., always present in real memory, especially during interrupts and other critical times) and, therefore, will never result in page faults when referenced.

If an area of pinned instruction memory is reordered, the area in the reordered text section where those instructions are moved must also be pinned. Since the granularity provided for pinning memory is usually at least a page, it can be quite inefficient to pin text reordered at the basic block level. One solution would be to pin the entire reordered text area. However, the base kernel usually has other position-dependent code that makes dynamic extension of the kernel more difficult than user-levelcode.

The solution developed and implemented here relies on the standard practice of building the AIX kernel with separate pinned and pageable sections. As illustrated in **Figure 6**, the kernel is built with a sufficiently large "hole" or reorder area in the pinned section; when TDPR is applied, all reordered text is moved to that pinned reorder area (TDPR target area). Through the use of this technique, reordered pinned code remains pinned and reordered pageable code becomes pinned.

Although one may argue that pinning code that was previously pageable reduces the effectiveness of a pageable kernel, a case can be made that reordered code, which is frequently executed code, should be pinned (or would be "paged-in" anyway) because of its utilization.

Debugging support

Reordering an executable program as described herein can impose some additional requirements in the area of program debugging. Any debugging information embedded in the executable file that points to code which has been reordered must be adjusted either in the executable file (if possible) or during the debug process. Also, AIX executables contain traceback entries at the end of every procedure which are used, among other things, to determine the procedure name for an instruction address if a program crash occurs. These traceback entries are not moved while reordering and are therefore not present in the reordered code (but are left intact relative to the instructions in the original text section).

Debugging a TDPR-reordered executable is possible by utilizing a special TDPR XCOFF section created in the reordered executable program file which provides a cross-reference table containing the original and reordered addresses for all moved code. Using this cross-reference information, along with the original text area which still has the traceback entries in place, the debugger (with minor modifications) can function as it would with the original program.

Conclusions

The application of trace-directed program restructuring on programs running in a hierarchical virtual memory system has the potential to produce significant performance enhancements and reductions in real memory requirements for both user-level and kernel programs. By using the prototype tools developed for this investigation, performance improvements for AIX executable programs of up to 73% and reductions in text real memory requirements of up to 61% were measured. For applications where the workloads are not critical to program behavior, producing a single reordered executable to realize these improvements should be feasible. In cases where different workloads change program behavior dramatically, providing multiple executables (each reordered for a specific workload type) or reordering for the most common workload may still prove beneficial.

The techniques described herein have been implemented in the IBM AIX software product "FDPR" (feedback directed program restructuring); preliminary results indicate significant performance improvements for a variety of programs.

Opportunities for additional work in this area include the development of "optimal" algorithms for reorder-list generation, including techniques to maintain pre-existing compiler optimizations and direct optimization for N-way set-associative cache collisions, multi-workload optimizations, and data reordering.

Acknowledgments

The author would like to thank Tom Keller for inspiring and supporting this work, Brian Twichell for his address trace tool, Bob Urquhart, William Alexander, Brian Twichell, Michael Fortin, Marc Stephenson, Bruce Mealey, and Maher Saba for their many fruitful discussions, Maurice Franklin and Ava Dixon for database testing, R. J. Rusnak for the hardware performance monitor, Robert Berry and Steve White for reviewing this manuscript, and Jerry Kilpatrick, Doug Matson, John Reysa, and Sohel Saiyed for their support and commitment.

References

1. Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," *Software—Pract. & Exper.* **21**, No. 12, 1301–1321 (December 1991).
2. David W. Wall, "Predicting Program Behavior Using Real or Estimated Profiles," *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, June 26–28, 1991, pp. 59–70.
3. Karl Pettis and Robert C. Hansen, "Profile Guided Code Positioning," *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 20–22, 1990, pp. 16–27.
4. Scott McFarling, "Program Optimization for Instruction Caches," *ASPLOS–III Proceedings: Third International Symposium on Architectural Support for Programming Languages and Operating Systems*, Boston, April 3–6, 1989, pp. 183–191.
5. D. Ferrari, "The Improvement of Program Behavior," *Computer* **9**, No. 11, 39–47 (November 1976).
6. D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory," *IBM Syst. J.* **10**, No. 3, 168–192 (1971).

Received September 17, 1993; accepted for publication February 18, 1994

Randall R. Heisch *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (HEISCH at AUSVM6).*

Mr. Heisch joined IBM in 1984 and has worked in several areas, including two IBM Fellowship Projects, where he was involved in RISC microcode emulation and high-performance multimedia, AIX PC Simulator/6000 development, and the IBM Voice Communications Adapter and I/O products development groups. He received the B.S. degree in electrical engineering in 1980 and the M.S. degree in engineering in 1991 from the University of Texas at Austin. Mr. Heisch is currently an advisory programmer in the Processor and System Performance group; he has interests in high-performance real-time operating systems/executives, embedded processors and controllers, fast prototyping, and simulation.