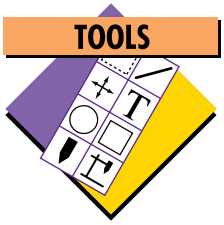


FDPR for AIX Executables

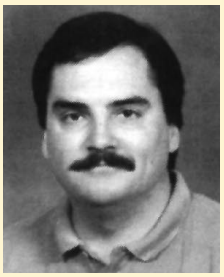
By Randall R. Heisch

The fdpr utility is a performance tuning tool used to optimize a program based on the actual hardware utilization characteristics of an application and its workload. The fdpr tool can improve both the execution time and real-memory utilization of user-level application programs by using a technique called Feedback Directed Program Restructuring (FDPR). Program speedups up to 73% and reductions in real-memory requirements up to 61% have been achieved using the FDPR reordering techniques.

The fdpr tool reorders and modifies the instructions in an AIX executable program by using profile information collected during the actual execution of the program. This improves instruction cache, instruction TLB (Translation Lookaside Buffer), and real-memory utilization by packing together highly executed code sequences and by recoding conditional branches to improve hardware branch prediction. It maximizes speedup by reordering instructions at the basic block level across the entire executable, independent of procedure or csect boundaries.

Highlights of the fdpr tool

- Program speedups up to 73% measured (typically 10-20%)
- Reductions in text memory requirements up to 61% (typically 20-30%)
- Global basic block-level instruction reordering
- Guaranteed functionality
- Debuggable with AIX 4.1 `dbx`



Randall R. Heisch

The AIX 4.1 `dbx` debug program has been modified to support executables reordered with the -R0 and -R2 options of fdpr. Additionally, the fdpr tool provides a reordering methodology[1] (option -R0) that guarantees functionality for reordered programs. The fdpr tool is part of the IBM Performance Aide 2.1 program product.

Why Reorder a Program?

Today's high-performance computer memory architectures are optimized for programs that exhibit high spatial 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 that result in processor designs optimized for those assumed characteristics (such as branching behavior and hardware branch prediction).

If these program assumptions are valid, processor performance is maximized. However, when a program deviates from these assumed characteristics, the processor architecture is inefficiently utilized. This often leads to reduced performance or excessive use of real memory.

Likewise, compilers attempt to generate optimum code for a specific hardware architecture on the basis of similar program assumptions. However, compiler optimizations are usually limited to a purely static analysis of a program, which includes speculation about how a program will execute on a given hardware platform. Since many programs result from binding together several separately compiled or assembled object modules, the compiler does not usually have an overall view of the final organization of the

[1] Heisch, R. R., "Trace-directed Program Restructuring for AIX Executables," *IBM Journal of Research and Development* 38, no. 4, (1994).

executable image. Therefore, it cannot perform a truly global optimization.

The fdpr tool 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 executing. It then uses that information to reorder and modify instructions across the entire executable program image to optimize use of hardware.

Figure 1 shows poor program locality for a typical high-level language code sequence. In this code sequence, the error path (taken when `x` equals `y`) is usually not executed (information not known at compile time).

Figure 2 shows the resulting assembler code generated for a code sequence of the form shown in Figure 1. The example represents a machine with 16 instructions per instruction-cache line. 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 reduced performance because of inefficient cache utilization. Similarly, real memory space may be wasted on instructions that are loaded into memory (because of their proximity to frequently executed code) but usually not executed.

Figure 2 also shows the results of reordering the instructions in the order they were executed. Based on information collected at runtime, the frequently executed code paths are grouped together. Additionally, the conditional branch instruction has been recoded to improve the success of hardware branch prediction. The result is twofold: a reduction in runtime memory requirements due to improved utilization of real memory pages, and improved performance due to reduced instruction cache and TLB miss rates and improved branch prediction.

Reordering frequently executed code sequences also results in 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 that results from these collisions. Reordering the instructions in a program based on the actual execution path can produce additional performance improvements by reducing the "conflict misses" in an N-way set-associative cache.

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

Figure 1. Example of poor locality of reference

Original Text			Reordered Text		
0x10000330	l	r2,0x14(r1)	0x10000330	b	0x10000590
0x10000334	l	r3,0x38(r1)	0x10000334	b	0x10000594
0x10000338	cmp	cr1,r2,r3	0x10000338	b	0x10000598
0x1000033c	bne	1,0x10000374	0x1000033c	bne	1,0x10000374
0x10000340	ai	r3,r31,0x8	0x10000340	ai	r3,r31,0x8
0x10000344	l	r4,0x38(r1)	.		
0x10000348	bl	0x10000530	.		
0x1000034c	l	r2,0x14(r1)			
0x10000350	ai	r3,r31,0x1c			
0x10000354	l	r4,0x38(r1)	0x10000590	l	r2,0x14(r1)
0x10000358	bl	0x10000530	0x10000594	l	r3,0x38(r1)
0x1000035c	l	r2,0x14(r1)	0x10000598	cmp	cr1,r2,r3
0x10000360	ai	r3,r31,0x30	0x1000059c	beq	1,0x10000340
0x10000364	l	r4,0x38(r1)	0x100005a0	l	r3,0x3c(r1)
0x10000368	bl	0x10000530	0x100005a4	l	r4,0x38(r1)
0x1000036c	l	r2,0x14(r1)	0x100005a8	bl	0x100005b0
0x10000370	b	0x10000384	0x100005ac	b	0x100005d8
0x10000374	l	r3,0x3c(r1)	0x100005b0	stu	r1,-64(r1)
0x10000378	l	r4,0x38(r1)	0x100005b4	st	r3,0x58(r1)
0x1000037c	bl	0x10000278	0x100005b8	st	r4,0x5c(r1)
0x10000380	st	r3,0x40(r1)	0x100005bc	l	r3,0x58(r1)
0x10000384	l	r3,0x38(r1)	0x100005c0	srai	r3,r3,0x3
			.		
			.		

Figure 2. Machine code examples for the source code shown in Figure 1

FDPR Process Overview

Figure 3 shows the overall process of applying FDPR. An execution profile is captured for the executable program to be reordered while it runs on the desired workload (W). The profile information is analyzed to determine an optimal instruction reordering. The reordering information from this analysis is then used to create a new, restructured executable. The reordered executable shows varying degrees of performance improvements or reduced instruction memory requirements when run on workload W (or similar workloads).

The fdpr tool follows a similar process and executes in three distinct phases.

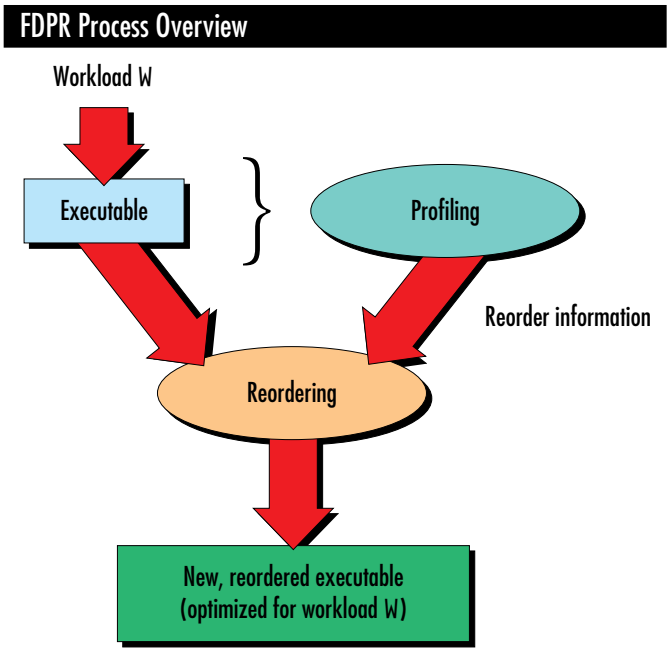


Figure 3. FDPR process overview

Program	POWER (8KIC)	POWER (32KIC)	POWER2	PowerPC 601 (0MB L2)
ksh	+45	+14	+13	+20
awk	+19	+9	+10	+11
vi	+13	+8	+6	+22
sed	+7	+5	+4	+6
SPEC 022.li	+20	+5	+4	+9
SPEC 072.sc	+11	+4	+1	+5
SPEC 056.ear	+9	+9	+4	+2
SPECint92'	-	-	+4[1]	+5[2]
RDBMS1 TPC-A	-	-	+15	-
RDBMS1 TPC-B	+17	+19	-	-
RDBMS2 TPC-C	-	+12	-	-

[1]66.7 MHz RS2G.9, 0 MB L2
[2]80 MHz PowerPC 601, 0 MB L2

Figure 4. Measured program speedups (%)

Phase 1: Creates an instrumented version of the executable
Phase 2: Runs the instrumented version to collect profile data
Phase 3: Reorders the original executable based on the profile data

Phase 1 builds a version of the executable to be reordered with the necessary instrumentation "hooks" required to collect the execution profile data. In phase 2, the instrumented executable created in the first phase is run for the desired workload (that is, used in some typical manner). In the last phase, the profile information is analyzed to determine an optimal instruction ordering, and the reordered version of the program is created. These phases can be run separately or in combination, but they must be in sequence.

In its simplest usage, the fdpr tool is invoked as follows:

```
fdpr  -p program  -x workload
```

where `program` specifies the name of the executable program to reorder, and `workload` specifies the desired invocation required to use that program. If successful, the reordered version of the program, `program.fdpr`, is created.

The fdpr tool provides four levels of optimization, that compare as follows:

- **R0/R2:** Guarantees functionality[2] (R0 only) of a reordered program; preserves debuggability at full program speedup; typically increases the size of the executable by 30-50%
- **R1/R3:** Maintains the size of the original executable; cannot guarantee functionality; marginally debuggable at the expense of speedup

Example fdpr Results

Figures 4 through 7 show typical results for reordering user-level applications. All results were collected using the fdpr -R2 optimization level. However, some results were measured during early prototype testing using an IBM internal-use-only trace tool for profiling.

Figure 4 shows the speedups measured for the 8KIC and 32KIC POWER, POWER2, and PowerPC 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. Note that each program shown in Figure 4 was reordered and tested on the same

[2] The -R0 option invokes a reordering methodology (see footnote 1) that can theoretically guarantee functionality; however, reordering may produce programs that behave unexpectedly. To verify expected functionality, programs reordered with the fdpr utility should be rigorously retested with at least the same test suite used for the original program.

workload specific to that test. Figure 7 presents the results for cross-workload measurements.

Source of the Improvements

Figure 5 shows the factors contributing to the 17% speedup measured for the RDBMS1 TPC-B test on the POWER machine. These measurements were taken using a hardware performance monitor that provides exact counts for clock cycles, instructions executed, instruction cache and TLB misses, and so on, during program execution.

The data indicates improved performance due to reduced instruction cache and TLB miss rates, and fewer conditionally issued instructions that were canceled (conditional branches that were predicted incorrectly).

Figure 6 shows the reductions in text real-memory requirements for several user-level applications. The changes in memory requirements were calculated using two different methods (shown as xx/yy). The first number (xx) represents the change in the total number of pages required for executing 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 that result in additional text memory pages touched during execution (a by-product of -R0 guaranteed functionality). However, the 61% reduction for the RDBMS1 TPC-B test represents an instruction memory savings of more than 512 KB for this program.

To achieve guaranteed functionality and preserve debuggability (using the fdpr -R0 or -R2 options), the reordered instructions are appended to the original executable. This results in increased file sizes for the executable programs as shown in Figure 6. However, for environments in which disk space is not extremely critical, the benefits of improved performance and reduced real-memory requirements more than compensate for the additional disk storage requirements.

Cross-Workload Effects

For certain applications, determining an appropriate workload for reordering a program can be a challenge. If two workloads execute a program differently, finding a single, optimal instruction ordering for both workloads is unlikely.

For example, a program is reordered for workload A; 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 workloads A and B together—in which 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 this reordered program on workload C, where workload C was not in the set of workloads used to reorder the program, typically yields little (or possibly negative) improvement, if workload C differs greatly from the other workloads.

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%

Figure 5. Factors contributing to RDBMS1 TPC-B speedup

Program Size	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

Figure 6. Text working set and executable size changes (%)

The cross-workload effects for two programs, `awk` and `ksh`, are shown in Figure 7. The reordered `awk` programs are as follows:

- `awk.heap`: Reordered for heapsort workload
- `awk.pts`: Reordered for an `awk` Performance Test Suite (PTS) workload
- `awk.comb`: Reordered for both the heapsort and PTS workloads

The reordered `ksh` programs are as follows:

- `ksh.scr`: Reordered for the `ksh` "built-in" commands workload `scr1`
- `ksh.sum`: Reordered for the sequential summation workload `sum.ksh`

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

Figure 7. Cross-workload results

- `ksh.comb`: Reordered for both `scr1` and `sum.ksh` workloads

As shown in Figure 7, running program `awk.pts` on the heapsort workload (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 when `awk.heap` is 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 similar to the `awk` results, with only a 3% speedup shown for `ksh.sum` running the `scr1` workload, and still significant speedups for `ksh.comb` on both workloads. The data suggests that careful selection of workloads is critical to achieve good overall speedups from reordering. However, to reach maximum performance improvements for `ksh`, `awk`, and these simple workloads, the program must be reordered for the exact workload for which it will be used.

One possible solution to the problem of cross-workload effects for widely varying workloads is to produce different versions of the program that are each optimized for specific workload types. Using this method, once the workload type is known, the executable program reordered for that specific workload is used.

Debugging Support

To achieve maximum program speedup, highly executed code sequences must be grouped together, independent of procedure or other structural boundaries. The "hot" code paths from multiple procedures or csects are reordered together, away from code that is not generally executed, as well as non-code such as traceback entries. However, a disadvantage of this blurring of procedure boundaries and removal of traceback entries is that program debug capability is reduced or completely disabled.

To solve this problem, the fdpr -R0 and -R2 options employ a unique reordering methodology that preserves debuggability, while requiring only minor modifications to the debugger. This methodology maintains the structure of the original program, which includes the relative location of traceback entries. Also, a new program section is created in the reordered program that contains a mapping of original to reordered addresses. Using this methodology, fdpr produces debuggable, reordered programs that simultaneously achieve maximum speedup by maintaining global instruction reordering.

Debug functionality is equivalent to that provided for any stripped or optimized executable. The AIX 4.1 `dbx` program will recognize and utilize the additional program section and the original program text area, and translate between reordered and original instruction addresses for program failures, stack traceback, and assembler single stepping.

Summary

The fdpr performance tuning utility enables improved performance and memory utilization by optimizing a program based on its actual behavior. Large programs with many conditional tests or highly structured programs with multiple, sparsely placed procedures offer the greatest potential for reordering improvements. Significant speedups have also been measured on smaller programs with poor hierarchical virtual memory utilization or branch behavior. For applications in which the workloads are not critical to program behavior, a single reordered executable can be produced to realize these benefits. For applications in which the workloads do change program behavior significantly, using fdpr to provide multiple executables (each reordered for a specific workload type) or reordering for the most common workload may still result in improvements.

Randall R. Heisch, IBM Corporation, RISC System/6000 Division, 11400 Burnet Road, Austin, TX 78758. Mr. Heisch is an advisory programmer in the Processor and System Performance group. He has a BS in Electrical Engineering and an MS in Engineering from the University of Texas at Austin.