Optimizing Alpha Executables on Windows NT with Spike

Robert S. Cohn
David W. Goodwin
P. Geoffrey Lowney

Many Windows NT–based applications are large, call-intensive programs, with loops that span multiple procedures and procedures that have complex control flow and contain numerous basic blocks. Spike is a profile-directed optimization system for Alpha executables that is designed to improve the performance of these applications. The Spike Optimizer performs code layout to improve instruction cache behavior and hot-cold optimization to reduce the number of instructions executed on the frequent paths through the program. The Spike Optimization Environment provides a complete system for performing profile feedback by handling the tasks of collecting, managing, and applying profile information. Spike speeds up program execution by as much as 33 percent and is being used to optimize applications developed by DIGITAL and other software vendors.

Spike is a performance tool developed by DIGITAL to optimize Alpha executables on the Windows NT operating system. This optimization system has two main components: the Spike Optimizer and the Spike Optimization Environment. The Spike Optimizer[1–3] reads in an executable, optimizes the code, and writes out the optimized version. The Optimizer uses profile feedback from previous runs of an application to guide its optimizations. Profile feedback is not commonly used in practice because it is difficult to collect, manage, and apply profile information. The Spike Optimization Environment[1] provides a user-transparent profile feedback system that solves most of these problems, allowing a user to easily optimize large applications composed of many executables and dynamic link libraries (DLLs).

Optimizing an executable image after it has been compiled and linked has several advantages. The Spike Optimizer can see the entire image and perform interprocedural optimizations, particularly with regard to code layout. The Optimizer can use profile feedback easily, because the executable that is profiled is the same executable that is optimized; no awkward mapping of profile data back to the source language takes place. Also, Spike can be used when the sources to an application are not available, which is beneficial when DIGITAL is working with independent software vendors (ISVs) to tune applications.

Applications can be loosely classified into two categories: loop-intensive programs and call-intensive programs. Conventional compiler technology is well suited to loop-intensive programs. The important loops in a program in this category are within a single procedure, which is typically the unit of compilation. The control flow is predictable, and the compiler can use simple heuristics to determine the frequently executed parts of the procedure.

Spike is designed for large, call-intensive programs; it uses interprocedural optimization and profile feedback. In call-intensive programs, the important loops span multiple procedures, and the loop bodies contain procedure calls. Consequently, optimizations on the loops must be interprocedural. The control flow is

complex, and profile feedback is required to accurately predict the frequently executed parts of a program. Call overhead is large for these programs. Optimizations to reduce call overhead are most effective with interprocedural information or profile feedback.

The Spike Optimizer implements two major optimizations to improve the performance of the call-intensive programs just described. The first is code layout:[4–6] Spike rearranges the code to improve locality and reduce the number of instruction cache misses. The second is hot-cold optimization (HCO):[7] Spike optimizes the frequent paths through a procedure at the expense of the infrequently executed paths. HCO is particularly effective in optimizing procedures with complex control flow and high procedure call overhead.

The Spike Optimization Environment provides a system for managing profile feedback optimization.[1] The user interface is simple—it requires only two user interactions: (1) the request to start feedback collection on an application and (2) the request to end collection and to use the feedback data to optimize the application. Spike maintains a database of profile information. When a user selects an application, Spike makes an entry in its database for the application and for each of its component images. For each image, Spike keeps an instrumented version, an optimized version, and profile information. When the original application is run, a transparency agent substitutes the instrumented or optimized version of the application, as appropriate.

This paper discusses the Spike performance tool and its use in optimizing Windows NT–based applications running on Alpha processors. In the following section, we describe the characteristics of Windows NT–based applications. Next, we discuss the optimizations used in the Spike Optimizer and evaluate their effectiveness. We then present the Spike Optimization Environment for managing profile feedback optimization. A summary of our results concludes the paper.

Characteristics of Windows NT–based Applications

To evaluate Spike, we selected applications that are typically used on Alpha computers running the Windows NT operating system. These applications include commercial databases, computer-aided design (CAD) programs, compilers, and personal productivity tools. For comparison, we also included the benchmark programs from the SPECint95 suite.[8] Table 1 identifies the applications and benchmarks, and the workloads used to exercise them. All programs are optimized versions of DIGITAL Alpha binaries and are compiled with the same highly optimizing back end that is used on the UNIX and OpenVMS systems.[9] The charts and graphs in this paper contain data from a core set of applications. Note that we do not have a full set of measurements for some applications.

In obtaining most of the profile-directed optimization results presented in this paper, we used the same input for both training and timing so that we could know the limits of our approach. Others in the field have shown that a reasonably chosen training input will yield reliable speedups for other input sets.[10] Our experience confirms this result. For the code layout results presented in Figure 11, we used the official SPEC timing harness to measure the SPECint benchmarks. This harness uses a SPEC training input for profile collection and a different reference input for timing runs.[8]

Figure 1 is a graph that shows, for each application and benchmark, the size of the single executable or DLL responsible for the majority of the execution time. The figure contains data for most of the applications and all the benchmarks listed in Table 1. Some Windows NT–based applications are very large. For example, PTC has 30 times more instructions than GCC, the largest SPECint95 benchmark. Large Windows NT–based applications have thousands of procedures and millions of basic blocks. With such programs, Spike achieves significant speedups by rearranging the code to reduce instruction cache misses. Code rearrangement should also reduce the working set of the program and the number of virtual memory page faults, although we have not measured this reduction.

To characterize a call-intensive application, we looked at SQLSERVR. We estimated the loop behavior of SQLSERVR by classifying each of its procedures by the average trip count of its most frequently executed loop, assigning a weight to each procedure based on the number of instructions executed in the procedure, and graphing the cumulative distribution of instructions executed. The graph is presented in Figure 2. Note that 69 percent of the execution time in SQLSERVR is spent in procedures that have loops with an average trip count less than 2. Nearly all the run time is spent in procedures with loops with an average trip count less than 16. An insignificant amount of time is spent in procedures containing loops with high trip counts. Of course, SQLSERVR executes many loops, but the loop bodies cross multiple procedures. To improve SQLSERVR performance, Spike uses code layout techniques to optimize code paths that cross multiple procedures. Also note that 69 percent of the execution time is spent in procedures where the entry basic block is the most frequently executed basic block. The entry basic block dominates the other blocks in the procedure, and compilers often find it a convenient location for placing instructions, such as register saves. In SQLSERVR, this placement is a poor decision. Our HCO targets this opportunity to

Table 1
Windows NT–based Applications for Alpha Processors and SPECint95 Benchmarks

Program	Full Name	Type	Workload
SQLSERVR	Microsoft SQL Server 6.5	Database	Transaction processing
SYBASE	Sybase SQL Server 11.5.1	Database	Transaction processing
EXCHANGE	Microsoft Exchange 4.0	Mail system	Mail processing
EXCEL	Microsoft Excel 5.0	Spreadsheet	BAPCo SYSmark for Windows NT Version 1.0
WINWORD	Microsoft Word 6.0	Word processing	BAPCo SYSmark for Windows NT Version 1.0
TEXIM	Welcom Software Technology Texim Project 2.0e	Project management	BAPCo SYSmark for Windows NT Version 1.0
MAXEDA	Orcad MaxEDA 6.0	Electronic CAD	BAPCo SYSmark for Windows NT Version 1.0
ACAD	Autodesk AutoCAD Release 13	Mechanical CAD	San Diego Users Group benchmark
CV	Computervision Pmodeler v6	Mechanical CAD	Mechanical model
PTC	Parametric Technology Corporation Pro/ENGINEER Release 18.0	Mechanical CAD	Bench97
SOLIDWORKS	SolidWorks Corporation SolidWorks 97	Mechanical CAD	Intake runner model
USTATION	Bentley Systems MicroStation 95	Mechanical CAD	Rendering
EDS	Electronic Data Systems Unigraphics 11.1	Mechanical CAD	Brake shoe model
MPEG	DIGITAL Light & Sound Pack	MPEG viewer	MPEG playback
C1, C2	Microsoft Visual C++ 5.0	Compiler C1: front end C2: back end	5,000 lines of C code
OPT, EM486	DIGITAL FX!32 Version 1.2	Emulation software OPT: x86-to-Alpha translator EM486: x86 emulator	BYTEmark benchmark
ESRI	Environmental Systems Research Institute ARC/INFO 7.1.1	Geographical Information Systems	Regional model
VORTEX	SPECint95	Database	SPEC reference
GO	SPECint95	Game	SPEC reference
M88KSIM	SPECint95	Simulator	SPEC reference
LI	SPECint95	LISP interpreter	SPEC reference
COMPRESS	SPECint95	Compression	SPEC reference
IJPEG	SPECint95	JPEG compression/ decompression	SPEC reference
GCC	SPECint95	C compiler	SPEC reference
PERL	SPECint95	Interpreter	SPEC reference

move instructions from the entry basic block to less frequently executed blocks.

Figure 3 presents the loop behavior data for many of the Windows NT–based applications listed in Table 1. Note that the applications fall into three groups. The most call-intensive applications are SQLSERVR, ACAD, and EXCEL, which spend approximately 70 percent of their run time in procedures with an average trip count less than 2. C2, WINWORD, and USTATION are moderately call intensive; they spend approximately 40 percent of their run time in loops with an average trip count less than 2. MAXEDA and TEXIM are loop intensive; they spend approximately 10 percent of their run time in loops with an average trip count less than 2. TEXIM is dominated by a single loop with an average trip count of 465.

We further characterized the nonlooping procedures by control flow. If a procedure consists of only a few basic blocks, techniques such as inlining are effective. To estimate the control flow complexity of

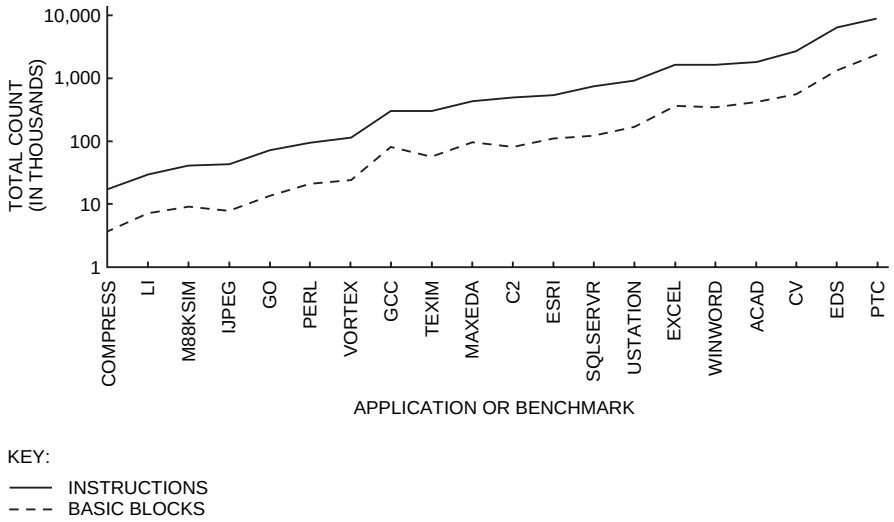


Figure 1
Size of Windows NT–based Applications and Benchmarks

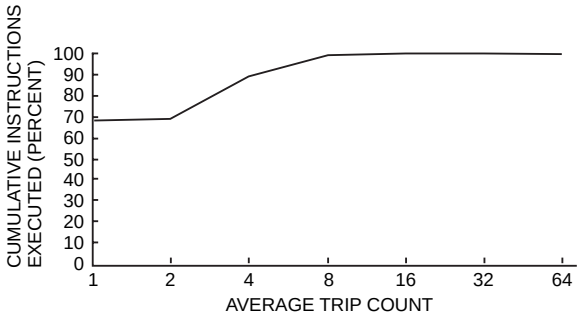


Figure 2
Loop Behavior of SQLSERVR

SQLSERVR, we classified each of its procedures by the number of basic blocks, assigned a weight to each procedure based on the number of instructions executed in the procedure, and graphed a cumulative distribution of the instructions executed. We restricted this analysis to procedures that have loops with an average trip count less than 4. (These procedures account for 69 percent of the execution time of SQLSERVR.) The line labeled ALL in Figure 4 represents the results of our analysis. Note that 90 percent of the run time of the nonlooping procedures is spent in procedures with more than 16 basic blocks. The line labeled FILTERED in Figure 4 represents the results when we ignored basic blocks that are rarely executed. Note that 65 percent of the run time of the nonlooping pro-

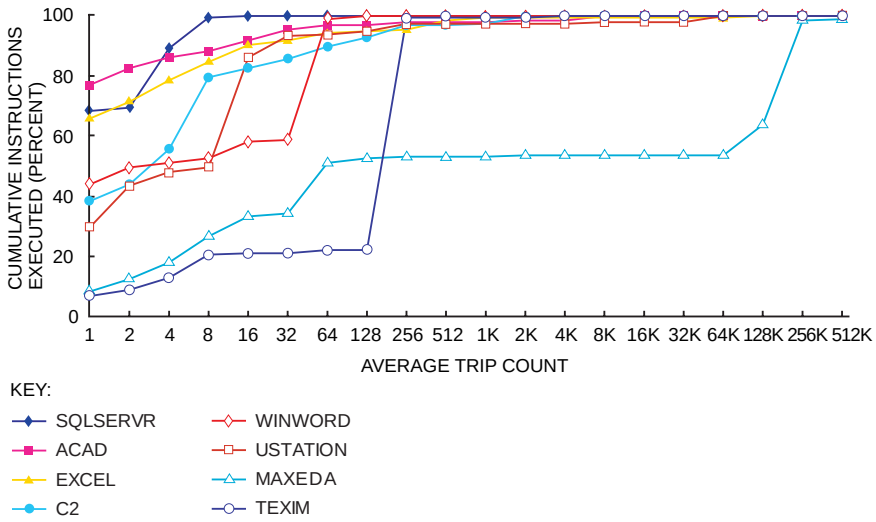


Figure 3
Loop Behavior of Windows NT–based Applications

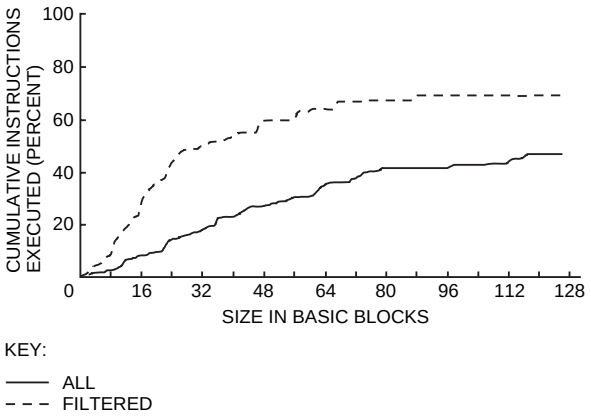


Figure 4
Complexity of Procedures in SQLSERVR for Procedures with an Average Trip Count Less Than 4, Which Account for 69 Percent of the Execution Time

cedures is spent in procedures with more than 16 basic blocks. In SQLSERVR, procedures are large; many basic blocks are executed, and many are not. Spike uses code layout and HCO to optimize the frequently executed paths through large procedures.

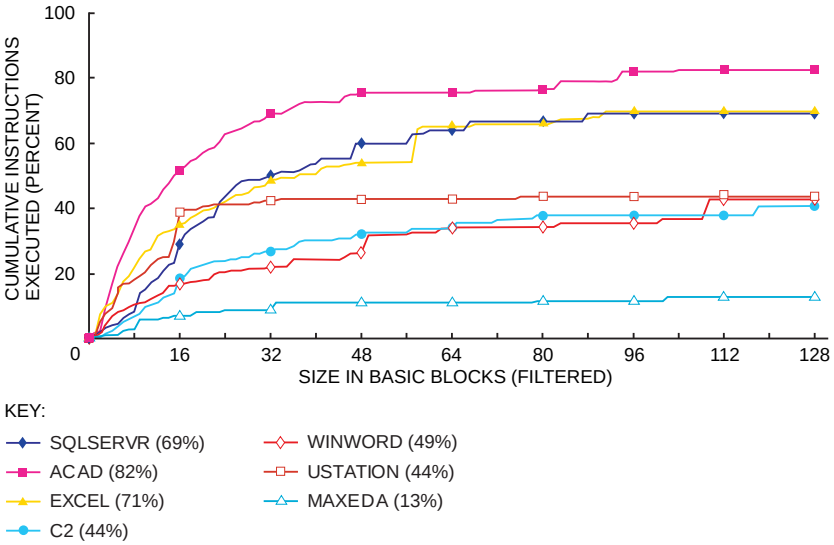
Figure 5 presents the control flow data for many of the Windows NT–based applications listed in Table 1. Again we measured only nonlooping procedures and ignored basic blocks that are rarely executed. Note that all the applications have large procedures. More than half the run time of the nonlooping procedures is spent in procedures that execute at least 16 basic blocks.

To estimate procedure call overhead, we counted the number of instructions executed in the prolog and epilog of each procedure. This estimate is conservative; it ignores the cost of the procedure linkage and argument setup and measures only the number of instructions used to create or remove a frame from the stack and to save or restore preserved registers. In SQLSERVR, 15 percent of all instructions are in prologs and epilogs. HCO removes approximately one half of this overhead.

The chart in Figure 6 shows the procedure call overhead for most of the Windows NT–based applications listed in Table 1. The overhead ranges from 23 percent to 2 percent. The applications are ordered according to the amount of run time in procedures with an average trip count less than 8 in Figure 3. The call overhead is roughly correlated with the amount of run time in low trip count procedures. Figure 6 includes data for some of the SPECint95 benchmarks, which are ordered by the amount of run time in procedures with an average trip count less than 2. The amount of call overhead for these benchmarks ranges from 24 percent to 0 percent and is more strongly correlated with the amount of run time in low trip count procedures.

Optimizations

The Spike Optimizer is organized like a compiler. It parses an executable into an intermediate representation, optimizes the representation, and writes out an optimized executable. The intermediate representation is a list of Alpha machine instructions, annotated



Note that the number that appears after the application name indicates the percentage of the total execution time spent in procedures with an average trip count less than 4.

Figure 5
Complexity of Procedures in Windows NT–based Applications for Procedures with an Average Trip Count Less Than 4

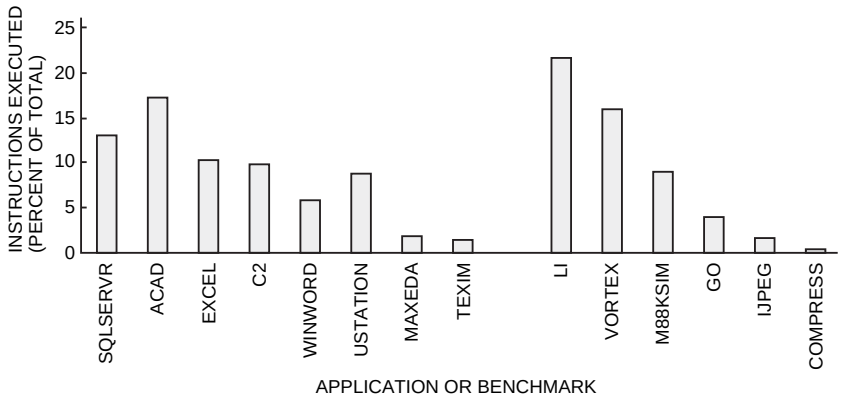


Figure 6
Procedure Call Overhead (Time Spent in Prolog and Epilog)

with a small amount of additional information. On top of the intermediate representation, the optimizer builds compiler-like structures, including basic blocks, procedures, a flow graph, a loop graph, and a call graph.[11] Images are large, and the algorithms and representations used in the optimizer must be time and space efficient.

The Spike Optimizer performs an interprocedural dataflow analysis to summarize register usage within the image.[12] This enables optimizations to use and reallocate registers. The interprocedural dataflow is fast, requiring less than 20 seconds on the largest applications we tested. Memory dataflow is much more difficult to analyze because of the limited information available in an executable, so the optimizer analyzes only references to the stack.

Optimizations rewrite the intermediate representation. The important optimizations are code layout and HCO. The Spike Optimizer also performs additional optimizations to reduce the overhead of shared libraries.

Code Layout

We derived our code layout algorithm from prior work on profile-guided code positioning by Pettis and Hansen.[6] The goal of the algorithm is to reduce instruction cache miss. Our algorithm consists of three steps. The first step reorganizes basic blocks so that the most frequent paths in a procedure are sequential, which permits more efficient use of cache lines and the exploitation of instruction prefetch. The second step places procedures in memory to avoid instruction cache conflicts. The third step splits procedures into hot and cold sections to improve the performance of procedure placement.

The following example illustrates basic block reorganization. Consider the flow graph in Figure 7, where each node is a basic block that contains four instructions. The arms of the conditional branches are labeled with their relative probabilities. Assume that the target is an Alpha 21164 processor.[13] Each instruction is 4 bytes, and the instruction cache is organized into 32-byte lines; each cache line holds two of the four-instruction basic blocks. A simple breadth-first code layout orders the code AB CD EF GH, and the common path ABDFGH requires four cache lines. Two cache lines (CD and EF) each contain a basic block that is infrequently used but which must be resident in the cache for the frequently used block to be executed. If we order the code so that the common path is adjacent (AB DF GH CE), the infrequently used blocks are in the same line (CE), and they do not need to be in the cache to execute the frequently used blocks.

Straight-line code is also better able to exploit instruction prefetch. On an instruction cache miss, the Alpha 21164 processor prefetches the next four cache lines into a refill buffer. After an instruction cache miss, the processor frequently is able to execute a straight-line code path without stalling if the code is in the second-level cache. A branch that is taken typically requires an additional cache miss if the target of the branch is not already in the instruction cache.

We reorganize the basic blocks using a simple, greedy algorithm, similar to the trace-picking algo-

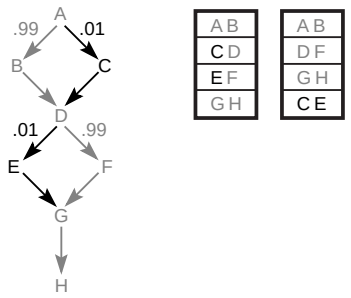


Figure 7
Basic Block Reorganization

rithm used in trace scheduling.[14] Our goal is to find a new ordering of the basic blocks so that the fall-through path is usually taken. We sort the list of flow graph edges by execution count and process them in order, beginning with the highest values. For each edge we make the destination basic block immediately follow the source block, unless the source has already been assigned a successor or the destination has already been assigned a predecessor.

We place procedures to avoid conflicts in the instruction cache. An Alpha 21164 has a primary instruction cache of 8 kilobytes (KB) that holds 256 lines of 32 bytes each. Two instructions conflict in the cache if they are more than 32 bytes apart and map to the same cache line, specifically, if *address*0/32 mod 256 = *address*1/32 mod 256. Our strategy is to place procedures so that frequently called procedures are near the caller. Consider the simple example in Figure 8. Assume procedure A calls procedure C in a loop. A and C map to the same cache lines, so on each call to C, C replaces A in the cache, and on each return from C, A replaces C. If we reorganize the code such that C follows A, both A and C can fit in the cache at once, and there are no conflict misses when A calls C.

We use another greedy algorithm to place procedures. The example presented in Figure 9 illustrates the steps. We build a call graph and assign a weight to each edge based on the number of calls. If there is more than one edge with the same source and destination, we compute the sum of the execution counts and delete all but one edge. Figure 9a shows the call graph. To place the procedures in the graph, we select the most heavily weighted edge (B to C), record that the two nodes should be placed adjacently, collapse the two nodes into one (B.C), and merge their edges (as shown in Figure 9b). We again select the most heavily weighted edge and continue (Figure 9c) until the graph is reduced to a single node A.D.B.C (Figure 9d). The final node contains an ordering of all the procedures. Special care is taken to ensure that we rarely require a branch to span more than the maximum branch displacement.

The effectiveness of procedure placement is limited by large procedures. In the PERL benchmark from SPEC, which is one of the smallest programs we studied, one frequently executed procedure is larger than 32 KB, four times the size of the instruction cache on the Alpha 21164 processor. In SQLSERVR, more than half the run time is spent in procedures with more than 16 basic blocks. To address this problem, we split procedures into hot and cold sections and treat each section as an independent procedure when placing procedures. To split a procedure, we examine each basic block and use a threshold on the execution count

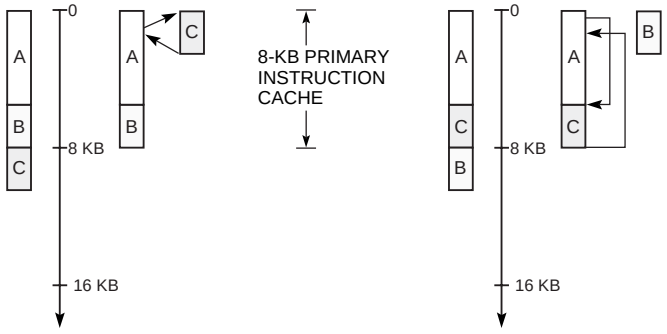


Figure 8
Procedure Placement

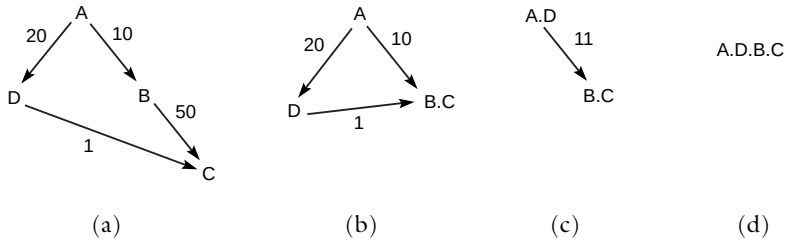
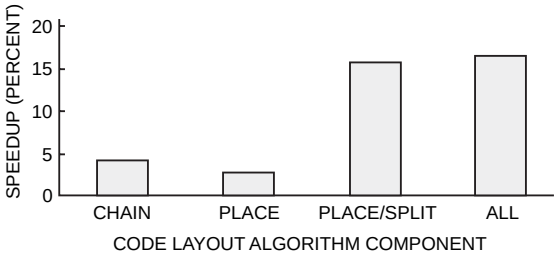


Figure 9
Steps in the Procedure Placement Algorithm

to decide if a basic block is cold. We use a single threshold for the entire program. The threshold is chosen so that the total execution time for all the basic blocks below the threshold constitutes no more than 1 percent of the execution time of the program. Procedures with both hot and cold basic blocks are split; otherwise, they are left intact.

Figure 10 illustrates the importance of procedure splitting. The figure charts the speedup on SQLSERVR, running on an Alpha 21064 workstation,[15] for the components of our code layout algorithm. The bar graph indicates that chaining basic blocks or placing procedures results in a speedup of less than 4 percent, but placing procedures after splitting yields a 15 percent speedup. Using all our optimizations (chaining, splitting, and placing) together produces a 16 percent speedup.

Figure 11 presents the speedups from code layout for the Windows NT–based applications and the SPECint benchmarks running on an Alpha 21164 workstation. Speedups range from 45 percent to 0 percent; most applications show a noticeable improvement. The leftmost seven Windows NT–based applications (SQLSERVR through TEXIM) are ordered by the amount of time spent in procedures with an average trip count less than 8 in Figure 3. Note that all but the most loop-intensive application show a significant speedup from code layout. Three programs show minimal speedup: TEXIM is dominated by a single loop that fits in the instruction cache, and IJPEG and COMPRESS are dominated by two or three small loops. These programs do not have an appreciable amount of instruction cache miss; changing the code layout cannot improve their performance.



Note that this data is for the SQLSERVR application running on an Alpha 21064 microprocessor.

Figure 10
Speedup for Code Layout by Optimization

Hot-Cold Optimization

Hot-cold optimization is a generalization of the procedure-splitting technique used in our code layout algorithm.[7] We optimize the hot part of the procedure (ignoring the cold part) by eliminating all instructions that are required only by the cold part. To implement this optimization, we create a hot procedure by copying the frequently executed basic blocks of a procedure. All calls to the original procedure are redirected to the hot procedure. Flow paths in the hot procedure that target basic blocks that were not copied are redirected to the appropriate basic block in the original (cold) procedure; that is, the flows jump into the middle of the original procedure. We then optimize the hot procedure, possibly at the expense of the flows that pass through the cold path.

HCO is best understood by working through an extended example. Consider the procedure `foo` (shown in Figure 12), which is a simplified version of a procedure from the Windows NT kernel.

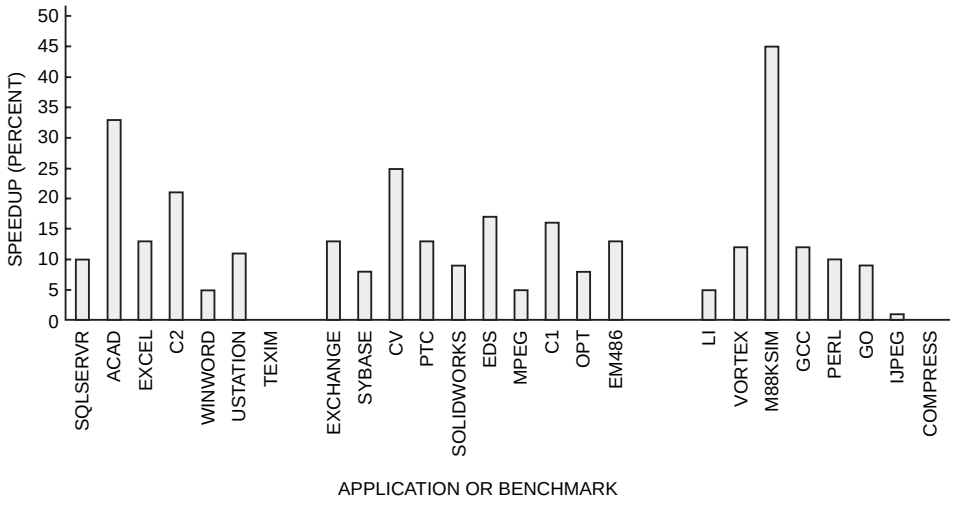


Figure 11
Speedup from Code Layout

```
 1  foo:     lda sp,16(sp)   ; adjust stack
 2           stq s0,0(sp)    ; save s0
 3           stq ra,8(sp)    ; save ra
 4           addl a0,1,s0    ; s0 = a0 + 1
 5           addl a0,a1,a0   ; a0 = a0 + a1
 6           bne s0,L2       ; branch if s0 != 0
 7      L1:  bsr f1          ; call f1
 8           addl s0,a0,t1   ; t1 = a0 + s0
 9           stl t1,40(gp)   ; store t1
10      L2:  ldq s0,0(sp)    ; restore s0
11           ldq ra,8(sp)    ; restore ra
12           lda sp,-16(sp)  ; adjust stack
13           ret (ra)        ; return
```

Figure 12
Simplified Version of a Procedure from the Windows NT Kernel

Assume that the branch in line 6 of `foo` is almost always taken and that lines 7 through 9 are almost never executed. When we copy the hot part of the procedure, we exclude lines 7 through 9 of `foo`. The resulting procedure `foo2` is shown in Figure 13.

```
 1    foo2:  lda sp,16(sp)
 2           stq s0,0(sp)
 3           stq ra,8(sp)
 4           addl a0,1,s0
 5           addl a0,a1,a0
 6           beq s0,L1
 7           ldq s0,0(sp)
 8           ldq ra,8(sp)
 9           lda sp,-16(sp)
10           ret (ra)
```

Figure 13
Hot Procedure

Note the reversal of the sense of the branch from `bne` in `foo` to `beq` in `foo2` and the change of the branch's target from `L2` to `L1`. All calls to `foo` are redirected to the hot procedure `foo2`. If the branch in line 6 of `foo2` is taken, then control transfers to line 7 of `foo`, which is in the middle of the original procedure. Once passed to the original procedure, control never passes back to the hot procedure. This feature of HCO enables optimization; when optimizing the hot procedure, we can relax some of the constraints imposed by the cold procedure.

So far, we have set up the hot procedure for optimization, but we have not made the procedure any faster. Now we show how to optimize the procedure. The hot procedure no longer contains a call, so we can delete the save and restore of the return address in lines 3 and 8 of `foo2` in Figure 13. If the branch transfers control to `L1` in the cold procedure `foo`, we must arrange for `ra` to be saved on the stack. In general, whenever we enter the original procedure from the hot procedure, we must fix up the state to match the expected state. We call the fix-up operations compensation code. To insert compensation code, we create a stub and redirect the branch in line 6 of `foo2` to branch to the stub. The stub saves `ra` on the stack and branches to `L1`.

Next, note that the instruction in line 5 of `foo2` writes `a0`, but the value of `a0` is never read in the hot procedure. `a0` is not truly dead, however, because it is still read if the branch in line 6 of `foo2` is taken. Therefore, we delete line 5 from the hot procedure and place a copy of the instruction on the stub. HCO tries to eliminate the uses of preserved registers in a procedure. Preserved registers can be more expensive than scratch registers because they must be saved and restored if they are used. Preserved registers are typically used when the lifetime of a value crosses a call. In the hot procedure, no lifetime crosses a call and the use of a preserved register is unnecessary. We rename all uses of `s0` in the hot procedure to use a free scratch register `t2`. We insert a copy on the stub from `t2` to `s0`. We can now eliminate the save and restore instructions in lines 2 and 7 of Figure 13 and place the save on the stub.

We have eliminated all references to the stack in the hot procedure. The stack adjusts on lines 1 and 9 in Figure 13 can be deleted from the hot procedure, and the initial stack adjust can be placed in the stub. The final code, including the stub `stub1`, is listed in Figure 14. The number of instructions executed in the frequent path has been reduced from 10 to 3. If the stub is taken, then the full 10 instructions and an extra copy and branch are executed.

```
1     foo2:  addl a0,1,t2
2            beq t2,stub1
3            ret (ra)
4     stub1: lda sp,16(sp)
5            stq s0,0(sp)
6            stq ra,8(sp)
7            addl a0,a1,a0
8            mov t2,s0
9            br L1
```

Figure 14
Optimized Hot Procedure

Finally, we would like to inline the hot procedure. Copies of instructions 1 and 2 can be placed inline. For the inlined branch, we must create a new stub that materializes the return address into `ra` before transferring control to `stub1`.

Except for partial inlining, we have implemented all the HCO optimizations in Spike. These optimizations are

- Partial dead code elimination[16]—the removal of dead code in the hot procedure
- Stack pointer adjust elimination—the removal of the stack adjusts in the hot procedure
- Preserved register elimination—the removal of the save and restore of preserved registers in the hot procedure

- Peephole optimization—the removal in the hot procedure of self-assignments and conditional branches with an always-false condition

Figure 15 shows coverage statistics for the HCO optimizations. Coverage represents the percentage of execution time spent in each category. To compute coverage, we assigned each procedure to a category and then for each category calculated the total number of instructions executed by its procedures. The category OPTIMIZED indicates the set of procedures optimized by HCO. The portion of the execution time spent in these procedures is typically 60 percent but often higher. The category INFREQUENT is the set of procedures whose execution times are so small (less than 0.1 percent of the total time) that we did not think it was worthwhile to optimize the procedures. Ignoring procedures with small execution times allows us to optimize less than 5 percent of the instructions in a program, a significant reduction in optimizer time. The category NO SPLIT represents the procedures that we could not split into hot and cold parts because all basic blocks had similar execution counts. The category SP MODIFIED contains procedures in which the stack pointer is modified after the initial stack adjust in the prolog. We decided not to optimize these procedures, but it is possible to do so with extra analysis. Note that the execution time spent in this category of procedures is small except for in C2, where the category contains two procedures and the coverage is 7 percent. Finally, the category NO ADVANTAGE represents the procedures that were split but that the optimizer was not able to improve.

Figure 16 shows the overall reduction in path length as a result of HCO, broken down by optimization. Most of the reduction in path length comes equally from the removal of unnecessary save and restore instructions and from the removal of partial dead code. Stack pointer adjust elimination and peephole optimization result in smaller additional gains. A large peephole category is usually the result of a save and restore of a preserved register that is made unnecessary by HCO; the restore is converted to a self-assignment by copy propagation, which is then removed by peephole optimization.

HCO is most effective on call-intensive programs such as SQLSERVR, ACAD, and C2, where we eliminate calls when creating the hot procedures. For WINWORD, the speedup is small because coverage is low; we could not find a way to split the procedures.

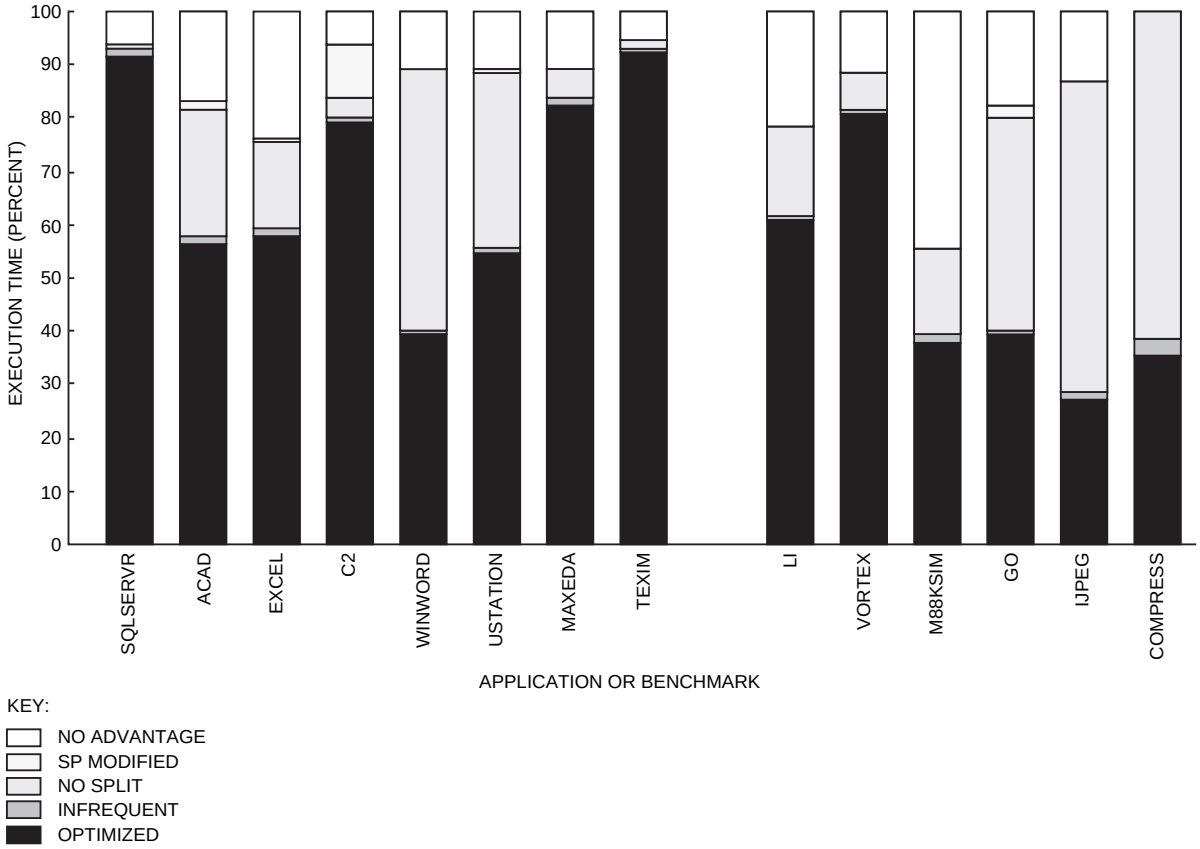


Figure 15
HCO Coverage by Execution Time

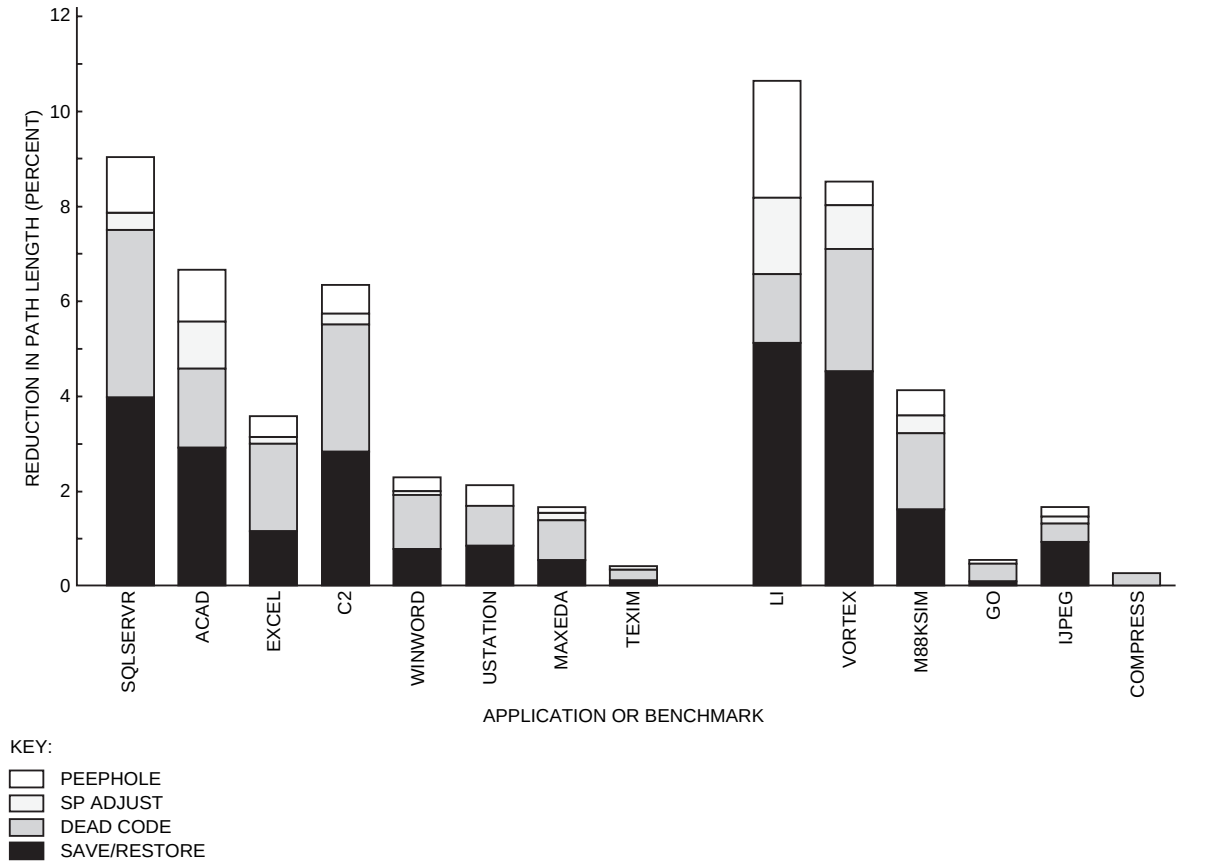


Figure 16
Reduction in Path Length As a Result of HCO

For EXCEL, HCO was able to split the procedures, but there is often a call in the hot path. Inlining may help in optimizing EXCEL, but frequently the call is to a shared library.

HCO is less effective on loop-intensive programs such as USTATION, MAXEDA, and TEXIM. HCO provides a framework for optimizing loops, and Chang, Mahlke, and Hwu have shown that eliminating the infrequent paths in loops enables additional optimizations, such as loop invariant removal.[17] However, our current implementation of Spike includes almost no information about the aliasing of memory operations; it can only optimize operations to local stack locations, such as spills of registers.

A leaf procedure is a procedure that does not contain a procedure call. Figure 17 compares the amount of time spent in leaf procedures before and after HCO is applied. By eliminating infrequent code, HCO is able to eliminate all calls in procedures that represent 10 percent to 20 percent of the execution time in C2, ACAD, SQLSERVR, and MAXEDA. For the other Windows NT–based applications, the increase in time spent in leaf procedures is very small. Most Windows NT–based applications spend much less than half the time in leaf procedures. To improve the performance of these applications, an optimizer needs to improve the performance of code with calls in the frequent path.

Code size and its effect on cache behavior is a major concern for us. In large applications, locality for instructions is present but not high. If an optimization decreases path length but also decreases locality as a side effect, the net result can be a loss in performance.

Figure 18 shows the total increase in code size as a result of optimization. HOT + COLD is the part of the increase that comes from replacing a single procedure with the original procedure plus a copy of the hot part. STUB is the increase attributed to stub procedures. Overall, the increase in size is small. The maximum increase is 11.6 percent for C2. SQLSERVR has the best speedup and is only 3.1 percent larger. Looking at the increase in total code size is misleading, however. HCO is not applied to procedures that are executed infrequently, which typically account for more than 95 percent of the instructions in a program, so tripling the size of optimized procedures would result in only a modest increase in code size. Note that tripling the size of the active part of an application usually disastrously decreases performance.

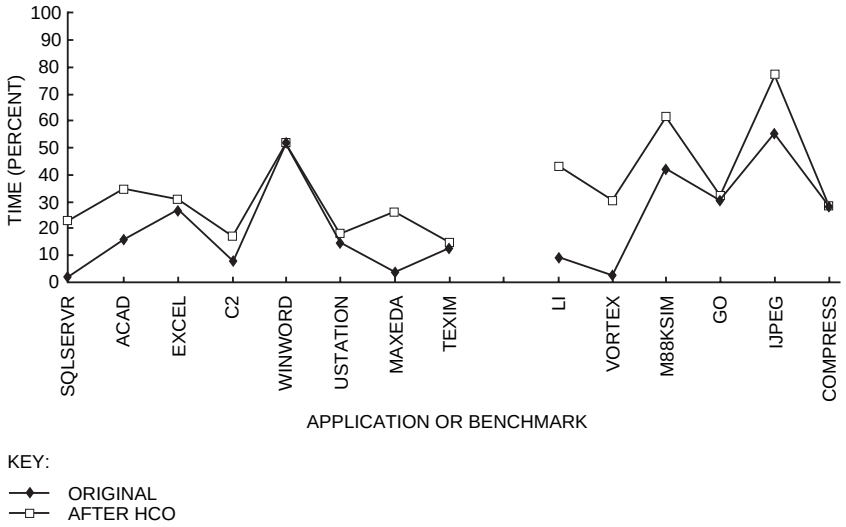


Figure 17
Time Spent in Leaf Procedures before and after HCO

For this reason, we also measured the increase in code size based on the procedures that were optimized. Figure 19 compares the total sizes of the hot procedures with the total sizes of the original procedures from which they were derived. For each procedure, by copying just the frequently executed part of the procedure, we excluded about 50 percent of the original. Next, we eliminated code that was frequently executed but only reachable through an infrequently executed path and therefore unreachable in the hot procedure. This code usually represents only 1 percent of the total size of a procedure. Finally, we optimized the hot procedure, reducing the remaining code size by about 10 percent, which is 5 percent of the size of the original procedure. The final sizes of the hot procedures as percentages of the sizes of the original procedures are shown in the line labeled HOT. Making the most frequently executed part of a program 50 percent to 80 percent smaller yields a big improvement in instruction cache behavior; however, it would be misleading to attribute this improvement to HCO, since our code layout optimization achieves the same result. When HCO is enabled, the cache layout optimizations are run after HCO. The baseline we compare against also has cache optimizations enabled, so improvements attributed to HCO are improvements beyond those that the other optimizations can make. HCO does make the frequently executed parts 10 percent

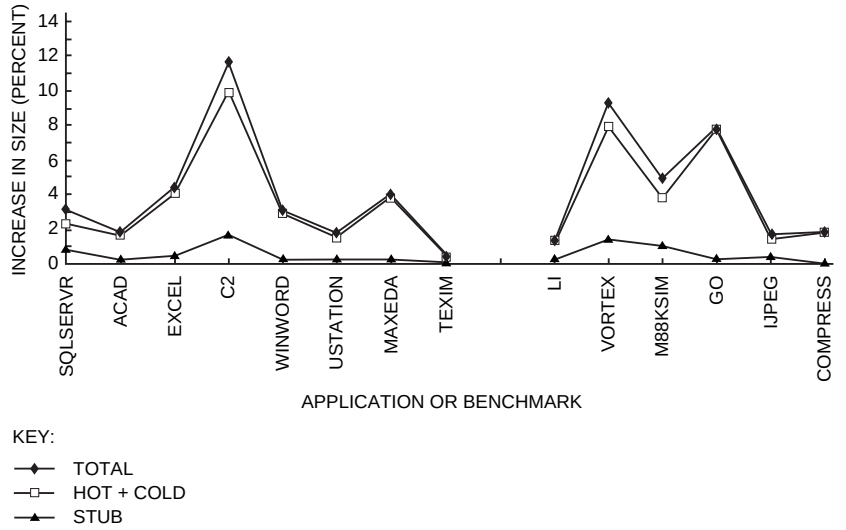


Figure 18
Overall Increase in Code Size after HCO

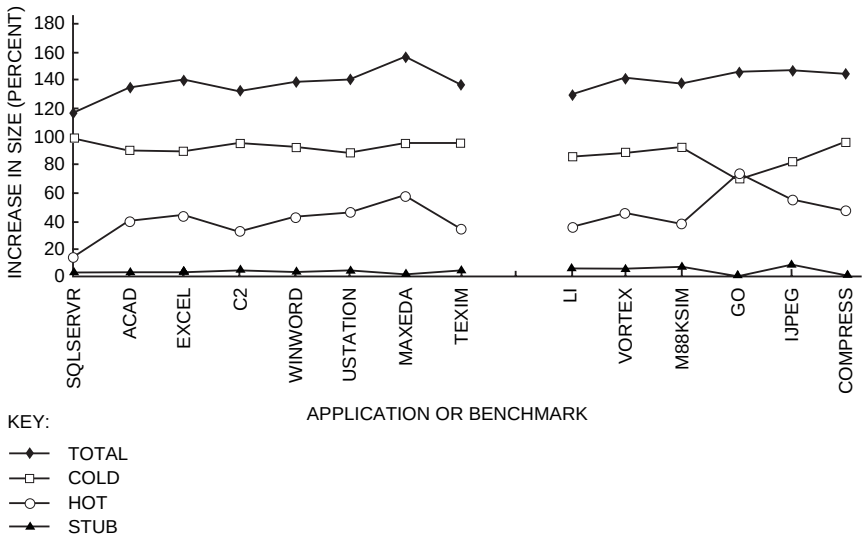


Figure 19
Size of Optimized Procedures after HCO

smaller, but we did not see significantly better instruction cache behavior when we ran programs with a cache simulator.

If we were to perform partial inlining, only the hot procedure would be copied. Since the hot procedure is less than half the size of the original procedure, partial inlining would greatly reduce the growth in code size due to inlining.

The line labeled COLD in Figure 19 shows how the size of the cold procedure is affected by HCO. When we redirect all calls to the hot procedure, some code in the original procedure becomes unreachable. The amount of unreachable code is usually less than 10 percent, which is much smaller than the 50 percent of the code we copied to create the hot procedure. The infrequent paths in a procedure often rejoin the frequent paths, which makes it necessary to have copies of both types of paths in the original procedure.

The line labeled STUB shows the code size of the stubs, which is very small. A stub contains the compensation code we introduce on a transition from a hot routine to a cold routine. We also implemented a variation of HCO that avoided stubs by reexecuting a procedure from the beginning instead of using a stub to reenter a routine in the middle. It is usually not possible to reexecute the procedure from the beginning because arguments have been overwritten. Given the small cost of stubs, we did not pursue this method.

The line labeled TOTAL shows that HCO makes the total code (HOT + COLD + STUB) 20 percent to 60 percent bigger. A procedure is partitioned so that there is less than a 1 percent chance that the stub and cold part are executed, so their size should not have a significant effect on cache behavior as long as the profile is representative.

Figure 20 shows how splitting affects the distribution of time spent among different procedure sizes for two programs where HCO is effective (C2 and SQLSERVR) and two programs where it is not (MAXEDA and WINWORD). For the graphs shown in parts a through d of Figure 20, we classified each procedure by its size in instructions before and after HCO and plotted two cumulative distributions of execution time. The farther apart the two lines, the better HCO was at shifting the distribution from large procedures to smaller procedures. Note that most of the programs spend a large percentage of the time in large procedures, which suggests that optimizers need to handle complex control flow well, even if profile information is used to eliminate infrequent paths.

Managing Profile Feedback Optimization

Profile feedback is rarely used in practice because of the difficulty of collecting, managing, and applying profile information. The Spike Optimization Environment[1] provides a system for managing profile feedback that simplifies this process.

The first step in profile-directed optimization is to instrument each image in an application so that when the application is run, profile information is collected. Instrumentation is most commonly done by using a compiler to insert counters into a program during compilation[18] or by using a post-link tool to insert counters into an image.[19,20] Statistical or sampling-based profiling is an alternative to counter-based techniques.[21,22] Some compiler-based and post-link systems require that the program be compiled specially, so that the resulting images are only useful for generating profiles. Many large applications have lengthy and

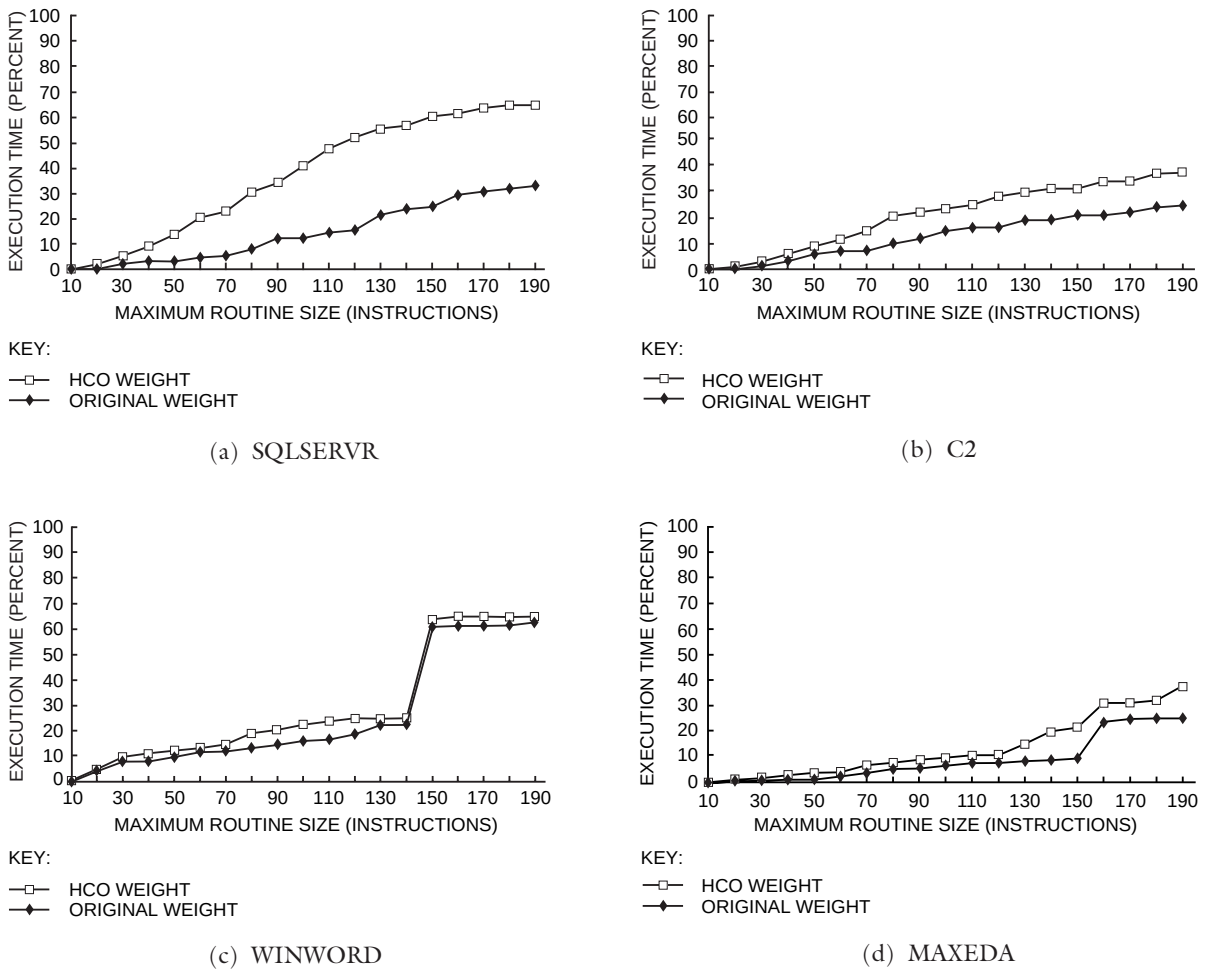


Figure 20
Cumulative Distribution of Execution Time by Procedure Size before and after HCO

complex build procedures. For these applications, requiring a special rebuild of the application to collect profiles is an obstacle to the use of profile-directed optimization.

Spike directly instruments the final production images so that a special compilation is not required. Spike does require that the images be linked to include relocation information; however, including this extra information does not increase the number of instructions in the image and does not prevent the compiler from performing full optimizations when generating the image.

Most applications consist of a main executable and many DLLs. Instrumenting all the images in an application can be difficult, especially when the user doing the profile-directed optimization does not know all the DLLs in the application. Spike relieves the user of this task by finding all the DLLs that the application uses, even if they are loaded dynamically with a call to LoadLibrary.

After instrumentation, the next step in profile-directed optimization is to execute the instrumented application and to collect profile information. Most profile-directed optimization systems require the user to first explicitly create instrumented copies of each image in an application. Then the user must assemble the instrumented images into a new version of the application and run it to collect profile information. As the profile information is generated, the user is responsible for locating all the profile information generated for each image and merging that information into a single set of profiles. Our experience with users has shown that requiring the user to manage the instrumented copies of the images and the profile information is a frequent source of problems. For example, the user may fail to instrument each image or may attempt to instrument an image that has already been instrumented. The user may be unable to locate all the generated profile information or may incorrectly combine the information. Spike frees the user

from these tedious and error-prone tasks by managing both the instrumented copy of each image and the profile information generated for the image.

After profile information is collected, the final step is to use the profile information to optimize each image. As with instrumentation, the typical profile-directed optimization system requires the user to optimize each image explicitly and to assemble the optimized application. Spike uses the profile information collected for each image to optimize all the images in an application and assembles the optimized application for the user.

Spike Optimization Environment

The Spike Optimization Environment (SOE) provides a simple means to instrument and optimize large applications that consist of many images. The SOE can be accessed through a graphical interface or through a command-line interface that provides identical functionality. The command-line interface allows the SOE to be used as part of a batch build system such as make.[23]

In addition to providing a simple-to-use interface, the SOE keeps the instrumented and optimized versions of each image and the profile information associated with each image in a database. When an application is instrumented or optimized, the original versions of the images in the application are not modified; instead, the SOE puts an instrumented or optimized version of each image into the database. When the user invokes the original version of an application, the SOE uses a transparency agent to execute the instrumented or optimized version.

The SOE allows the user to instrument and optimize an entire application using the following procedure:

1. Register: The user selects the application or applications that are to be instrumented and optimized. The user needs to specify only the application's main image. Spike then finds all the implicitly linked images (DLLs loaded when the main image is loaded) and registers that they are part of the application.
2. Instrument: The user requests that an application be instrumented. For each image in the application, the SOE invokes the Spike Optimizer to instrument that image. The SOE places the instrumented version of each image in the database. The original images are not modified.
3. Collect profile information: The user runs the original application in the normal way, e.g., from a command prompt, from Windows Explorer, or indirectly through another program. Our transparency agent (explained later in this section) invokes the instrumented version of the application in place of the original version. Any images dynamically loaded by the application are instrumented on the fly. Each time the application terminates, profile information for each image is written to the database and merged with any existing profile information.
4. Optimize: The user requests that an application be optimized. For each image in the application, the SOE invokes the Spike Optimizer to optimize the image using the collected profile information and places the optimized version of each image in the database.
5. Run the optimized version: The user runs the original application, and our transparency agent substitutes the optimized version, allowing the user to evaluate the effectiveness of the optimization.
6. Export: The SOE exports the optimized images from the database, placing them in a directory specified by the user. The optimized images can then be packaged with other application components.

The Spike Manager is the principal user interface for the SOE. The Spike Manager displays the contents of the database, showing the applications registered with Spike, the images contained in each application, and the profile information collected for each image. The Spike Manager enables the user to control many aspects of the instrumentation and optimization process, including specifying which images are to be instrumented and optimized and which version of the application is to be executed when the original application is invoked.

Transparent Application Substitution (TAS) is the transparency agent developed for the Spike system to execute a modified version of an application transparently, without replacing the original images on disk. TAS was modeled after the transparency agent in the DIGITAL FX!32 system[24] but uses different mechanisms. When the user invokes the original application, the SOE uses TAS to load an instrumented or optimized version. With TAS, the user does not need to do anything special to execute the instrumented or optimized version of an application. The user simply invokes the original application in the usual way (e.g., from a command prompt, from Windows Explorer, or indirectly through another application), and the instrumented or optimized application runs in its place. TAS performs application substitution in two parts. First, TAS makes the Windows NT loader use a modified version of the main image and DLLs. Second, TAS makes it appear to the application that the original images were invoked.

TAS uses debugging capabilities provided by the Windows NT operating system to specify that whenever the main image of an application is invoked, the modified version of that image should be executed instead. In each image, the table of imported DLLs is altered so that instead of loading the DLLs specified in the original image, each image loads its modified counterparts. Thus, when the user invokes an application, the Windows NT operating system loads the modified versions of the images contained in the application. Some applications load DLLs with explicit calls

to LoadLibrary. TAS intercepts those calls and instead loads the modified versions.

The second part of TAS makes the modified version of the application appear to be the original version of the application. Applications often use the name of the main image to find other files. For example, if an instrumented image requests its full path name, TAS instead returns the full path name of the corresponding original image. To do this, TAS replaces certain calls to kernel32.dll in the instrumented and optimized images with calls to hook procedures. Each hook procedure determines the outcome the call would have had for the original application and returns that result.

Instrumentation

Spike instruments an image by inserting counters into it. Using the results of these counters, the optimizer can determine the number of times each basic block and control flow edge in the image is executed. Spike uses a spanning-tree technique proposed by Knuth[25] to reduce the number of counters required to fully instrument an image. For example, in an if-then-else clause, counting the number of times the if and then statements are executed is enough to determine the number of times the else statement is executed. Register usage information is used to find free registers for the instrumentation code, thereby reducing the number of saves and restores necessary to free up registers.[12] Typically, instrumentation makes the code 30 percent larger. As part of the profile, Spike also captures the last target of a jump or procedure call that cannot be determined statically.

Spike's profile information is persistent; small changes to an image do not invalidate the profile information collected for that image. Profile persistence is essential for applications that require a lengthy or cumbersome process to generate a profile, even when using low-cost methods like statistical sampling. For example, generating a good profile of a transaction processing system requires extensive staging of the system. Even when it is possible to automate the generation of profiles, some ISVs find the extra build time unacceptable. With persistence, the user can collect a profile once and continue to use it for successive builds of a program as small changes are made to it. Our experience with an ISV has shown that the speedup from Spike declines as the profile gets older, but using a two- or three-week-old profile is acceptable. It is also possible to merge a profile generated by an older image with a profile generated by a newer image.

When using an old profile, Spike must match up procedures in the current program with procedures in the profiled program. Spike discards profiles for procedures that have changed. Relying on a procedure name derived from debug information to do the matching is not practical in a production environment. Instead, Spike generates a signature based on the flow graph of each procedure. Since signatures are not based on the code, small changes to a procedure will not invalidate the profile. Signatures are not unique, however, so it can be difficult to match two lists of signatures when there are differences. A minimum edit distance algorithm[26] is used to find the best match between the list of signatures of the current program and the list of signatures of the profiled program.

Summary

Many Windows NT–based applications are large, call-intensive programs, with loops that cross multiple procedures and procedures that have complicated control flow and many basic blocks. The Spike optimization system uses code layout and hot-cold optimization to optimize call-intensive programs. Code layout places the frequently executed portions of the program together in memory, thereby reducing instruction cache miss and improving performance up to 33 percent. Our code layout algorithm rearranges basic blocks so that the fall-through path is the common path. The algorithm also splits each procedure into a frequently executed (hot) part and an infrequently executed (cold) part. The split procedures are placed so that frequent (caller, callee) pairs are adjacent.

The hot part of a procedure is the collection of the common paths through the procedure. These paths can be optimized at the expense of the cold paths by removing instructions that are required only if the cold paths are executed. Hot-cold optimization exploits this opportunity by performing optimizations that remove partially dead code and replace uses of preserved registers with uses of scratch registers. Hot-cold optimization reduces the instruction path length through the call-intensive programs by 3 percent to 8 percent.

Profile feedback is rarely used because of the difficulty of collecting, managing, and applying profile information. Spike provides a complete system for profile feedback optimization that eliminates these problems. It is a practical system that is being actively used to optimize applications for Alpha processors running the Windows NT operating system.

Acknowledgments

Tryggve Fossum supported the development of Spike from the beginning; he also implemented two of our early optimizations. David Wall helped us get started parsing Windows NT images. Mike McCallig implemented our first symbol table package. Norman Rubin contributed to the design of the transparency agent. Many people helped collect the data presented in this paper, including Michelle

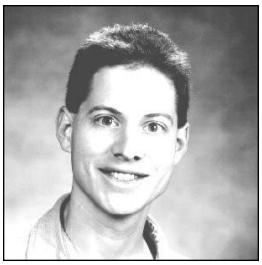
Alexander, Brush Bradley, Bob Corrigan, Jeff Donsbach, Hans Graves, John Henning, Phil Hutchinson, Herb Lane, Matt Lapine, Wei Liu, Jeff Seltzer, Arnaud Sergent, John Shakshober, and Robert Zhu.

References

1. R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, "Spike: An Optimizer for Alpha/NT Executables," *The USENIX Windows NT Workshop Proceedings,* Seattle, Wash. (August 1997): 17–24.
2. A. Srivastava and D. Wall, "Link-time Optimization of Address Calculation on a 64-bit Architecture," *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation,* Orlando, Fla. (June 1994): 49–60.
3. L. Wilson, C. Neth, and M. Rickabaugh, "Delivering Binary Object Modification Tools for Program Analysis and Optimization," *Digital Technical Journal,* vol. 8, no. 1 (1996): 18–31.
4. S. McFarling, "Program Optimization for Instruction Caches," *ASPLOS III Proceedings,* Boston, Mass. (April 1989): 183–193.
5. W. Hwu and P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proceedings of the Sixteenth Annual International Symposium on Computer Architecture,* Jerusalem, Israel (June 1989).
6. K. Pettis and R. Hansen, "Profile Guided Code Positioning," *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation,* White Plains, N.Y. (June 1990): 16–27.
7. R. Cohn and P. G. Lowney, "Hot Cold Optimization of Large Windows/NT Applications," *MICRO-29,* Paris, France (December 1996): 80–89.
8. Information about the SPEC benchmarks is available from the Standard Performance Evaluation Corporation at http://www.specbench.org/.
9. D. Blickstein et al., "The GEM Optimizing Compiler System," *Digital Technical Journal,* vol. 4, no. 4 (1992): 121–136.
10. B. Calder, D. Grunwald, and A. Srivastava, "The Predictability of Branches in Libraries," *Proceedings of the Twenty-eighth Annual International Symposium on Microarchitecture,* Ann Arbor, Mich. (November 1995): 24–34.
11. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools* (Reading, Mass.: Addison-Wesley, 1985).
12. D. Goodwin, "Interprocedural Dataflow Analysis in an Executable Optimizer," *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation,* Las Vegas, Nev. (June 1997): 122–133.
13. *Alpha 21164 Microprocessor Hardware Reference Manual,* Order No. EC-QAEQB-TE (Maynard, Mass.: Digital Equipment Corporation, April 1995).
14. J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers,* C-30, 7 (July 1981): 478–490.
15. *DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors Hardware Reference Manual,* Order No. EC-Q9ZUA-TE (Maynard, Mass.: Digital Equipment Corporation, June 1994).
16. J. Knoop, O. Rüthing, and B. Steffen, "Partial Dead Code Elimination," *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation,* Orlando, Fla. (June 1994): 147–158.
17. P. Chang, S. Mahlke, and W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," *Software—Practice and Experience,* vol. 21, no. 12 (1991): 1301–1321.
18. P. G. Lowney et al., "The Multiflow Trace Scheduling Compiler," *The Journal of Supercomputing,* vol. 7, no. 1/2 (1993): 51–142.
19. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation,* Orlando, Fla. (June 1994): 196–205.
20. *UMIPS-V Reference Manual (pixie and pixstats)* (Sunnyvale, Calif.: MIPS Computer Systems, 1990).
21. J. Anderson et al., "Continuous Profiling: Where Have All the Cycles Gone?" *Proceedings of the Sixteenth ACM Symposium on Operating System Principles,* Saint-Malo, France (October 1997): 1–14.
22. X. Zhang et al., "System Support for Automatic Profiling and Optimization," *Proceedings of the Sixteenth ACM Symposium on Operating System Principles,* Saint-Malo, France (October 1997): 15–26.
23. S. Feldman, "Make—A Program for Maintaining Computer Programs," *Software—Practice and Experience,* vol. 9, no. 4 (1979): 255–265.
24. R. Hookway and M. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal,* vol. 9, no. 1 (1997): 3–12.
25. D. Knuth, *The Art of Computer Programming: Vol. 1, Fundamental Algorithms* (Reading, Mass.: Addison-Wesley, 1973).
26. W. Miller and E. Meyers, "A File Comparison Program," *Software—Practice and Experience,* vol. 11 (1985): 1025–1040.

Biographies

Robert S. Cohn
Robert Cohn is a consulting engineer in the VSSAD Group, where he works on advanced compiler technology for Alpha microprocessors. Since joining DIGITAL in 1992, Robert has implemented profile-feedback and trace scheduling in the GEM compiler. He also implemented the code layout optimizations in UNIX OM. Robert has been a key contributor to Spike, implementing both hot-cold optimization and the code layout optimizations. Robert received a B.A. from Cornell University and a Ph.D. in computer science from Carnegie Mellon University.



David W. Goodwin
David W. Goodwin is a principal engineer in the VSSAD Group, where he works on architecture and compiler advanced development. Since joining DIGITAL in 1996, he has contributed to the performance analysis of the 21164, 21164PC, and 21264 microprocessors. For the Spike project, David implemented the Spike Optimization Environment and the interprocedural dataflow analysis. David received a B.S.E.E. from Virginia Tech. and a Ph.D. in computer science from the University of California, Davis.

P. Geoffrey Lowney
P. Geoffrey Lowney is a senior consulting engineer in the VSSAD Group, where he works on compilers and architecture to improve the performance of Alpha microprocessors. Geoff is the leader of the Spike project. For Spike, he implemented the infrastructure for parsing executables. Prior to joining DIGITAL in 1991, Geoff worked at Hewlett Packard/Apollo, Multiflow Computer, and New York University. Geoff received a B.A. in mathematics and a Ph.D. in computer science, both from Yale University.