Digital Technical Journal

digital™

Volume 9 Number 4
1997

Cover Design
The power of graphics workstations is measured by the speed at which the machine can create and manipulate 3-D objects. The PowerStorm 4D60T graphics adapter design, a topic featured in this issue, combines Alpha 64-bit microprocessor technology and modified rendering technology to attain high levels of performance both on leading real-world CAD/CAM applications such as Pro/ENGINEER, and on widely accepted industry benchmarks such as OpenGL Viewperf. Our cover design is made up of representative images from the Viewperf benchmark program and standard viewsets.

The cover was designed by Lucinda O'Neill of the DIGITAL Industrial and Graphic Design Group. The editors thank author Benjamin Lipchak for supplying the images used on the cover.

The *Digital Technical Journal* is a refereed journal published quarterly by Digital Equipment Corporation, 50 Nagog Park, AKO2-3/B3, Acton, MA 01720-9843.

Hard-copy subscriptions can be ordered by sending a check in U.S. funds (made payable to Digital Equipment Corporation) to the published-by address. General subscription rates are $40.00 (non-U.S. $60) for four issues and $75.00 (non-U.S. $115) for eight issues. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. DIGITAL customers may qualify for gift subscriptions and are encouraged to contact their sales representatives.

Electronic subscriptions are available at no charge by accessing URL **http://www.digital.com/subscription**. This service will send an electronic mail notification when a new issue is available on the Internet.

Single copies and back issues can be ordered by sending the requested issue's volume and number and a check for $16.00 (non-U.S. $18) each to the published-by address. Recent issues are also available on the Internet at **http://www.digital.com/dtj**.

DIGITAL employees may order subscriptions through Readers Choice at URL **http://webrc.das.dec.com**.

Inquiries, address changes, and complimentary subscription orders can be sent to the *Digital Technical Journal* at the published-by address or the electronic mail address, **dtj@digital.com**. Inquiries can also be made by calling the *Journal* office at 978-264-7556.

Comments on the content of any paper and requests to contact authors are welcomed and may be sent to the managing editor at the published-by or electronic mail address.

ISSN 0898-901X

Documentation Number EC-P9257-18

Book production was done by Quantic Communications, Inc.

Contents

Editor's Introduction

In 1992, DIGITAL announced the fastest 64-bit RISC microprocessor, the Alpha, with a clock rate of 200 MHz. Today's Alpha processor remains the leader in performance; the newest generation operates at 600 MHz, and the next generation will operate at greater than 1,000 MHz — gigahertz speed. With the industry's most powerful processor in hand, DIGITAL's engineers are working to apply Alpha in different areas of computing and effect optimal solutions to computing problems. Samples of that work are presented in this issue and include programming performance tools, the OpenVMS operating system for very large memory (VLM) applications, graphics adapters for workstations, and the DART network adapter for high-end systems.

Spike is a profile-directed performance tool for optimizing Alpha executables running on the Windows NT operating system. Designed specifically to improve the performance of large, call-intensive programs, such as commercial databases, CAD programs, compilers, and productivity tools, Spike has been shown to speed program execution by as much as 33 percent. Robert Cohn, Dave Goodwin, and Geoff Lowney describe Spike's two components. The Optimizer modifies code layout to improve instruction cache behavior and performs hot-cold optimization to reduce the number of instructions executed on frequent paths through the program. The Optimization Environment collects, manages, and applies profile information transparently for the programmer.

An experimental Atom-based performance tool presented by Susanne Balle and Simon Steely provides programmers with an understanding of the access pattern behavior of their technical applications. The tool generates histograms for each memory reference in a program, thus allowing the programmer to spot bottlenecks. The authors step through an instructive case study in the use of the tool with Fortran programs, showing how different compiler switches affect the execution of a program algorithm.

The OpenVMS Alpha operating system version 7.1 extends its support for VLM applications. The design work discussed by Karen Noel and Nitin Karkhanis focused on increasing flexibility for VLM applications and on adding system management capabilities. Areas reviewed are the shared memory objects designed to improve application scaling on the system, shared page tables to reduce application start-up/shut-down times, and the physical memory reservation system to allow efficient application use of system components, namely the translation buffer.

DIGITAL's PowerStorm series of graphics adapters for mid-range workstations provides exceptional performance on the DIGITAL UNIX and the Windows NT operating systems. Benj Lipchak, Tom Frisinger, Karen Bircsak, Keith Comeford, and Mike Rosenblum have written an informative tutorial about the PowerStorm adapter design that was shaped in large part by the existing competitive environment. Their discussion covers selected benchmarks and real-world performance experiences, the advantages and disadvantages in choosing a direct-rendering or an indirect-rendering scheme, and the ways in which the engineering team exploited the Alpha microprocessor's exceptional floating-point speed.

DART is a 622-megabit-per-second network adapter that connects gigabit-class networks to gigabit-class I/O buses. It is designed to increase network throughput and decrease system overhead. Bob Walsh explains that the DART project, started in the late 1980s, anticipated the need to address fundamental memory bandwidth bottleneck issues from a system-level perspective. The main approach taken in the DART adapter is data copy avoidance, without requiring changes to system call semantics.

The upcoming *Journal* will be a special issue that features papers on programming languages and tools. Topics include C and Fortran parallelizing compilers, the C++ template facility, alias analysis algorithms, debuggers, and performance tools for software running on the Windows NT, UNIX, and OpenVMS operating systems.

Jane Blake

Jane C. Blake
Managing Editor

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

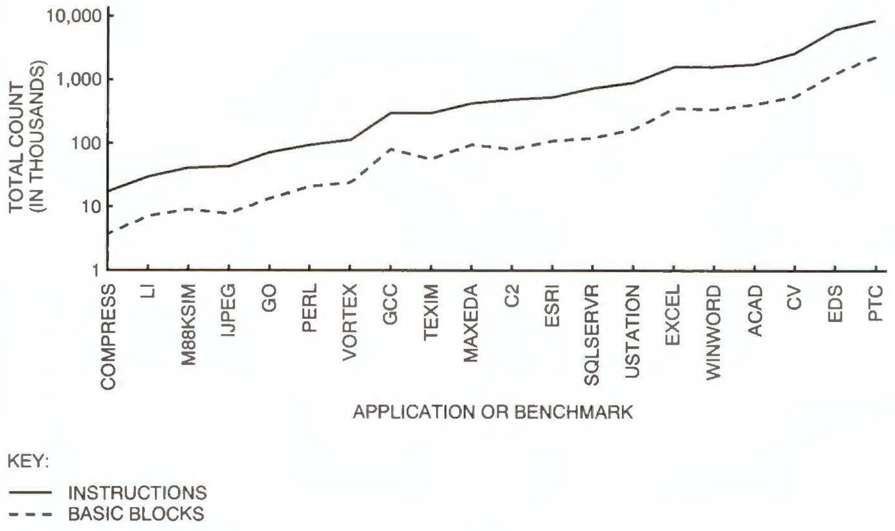


Figure 1
Size of Windows NT–based Applications and Benchmarks

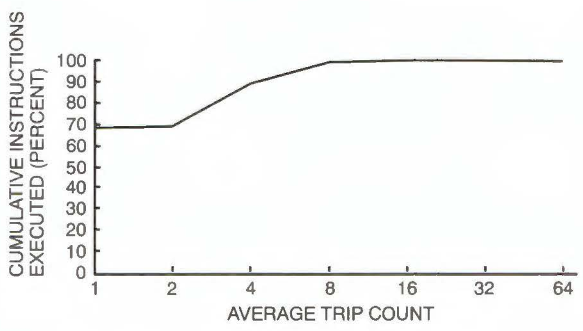


Figure 2
Loop Behavior of SQLSERVR

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-

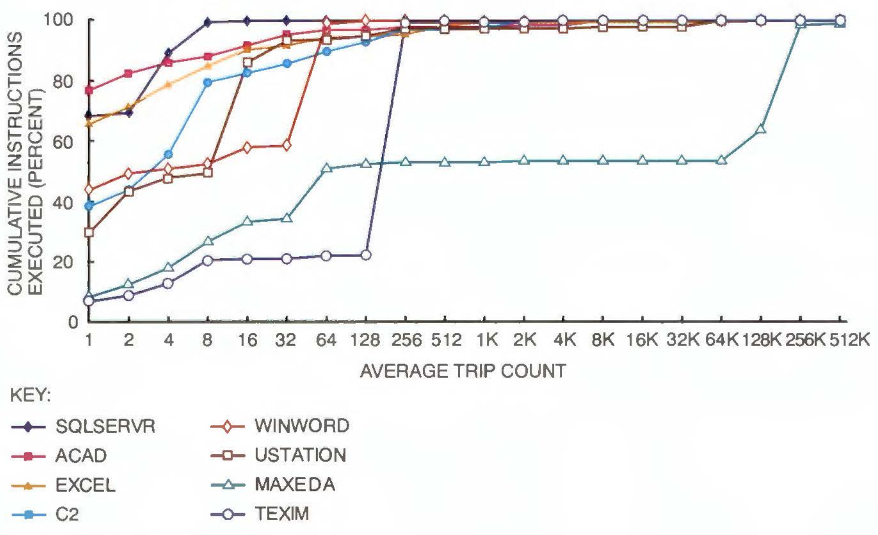


Figure 3
Loop Behavior of Windows NT–based Applications

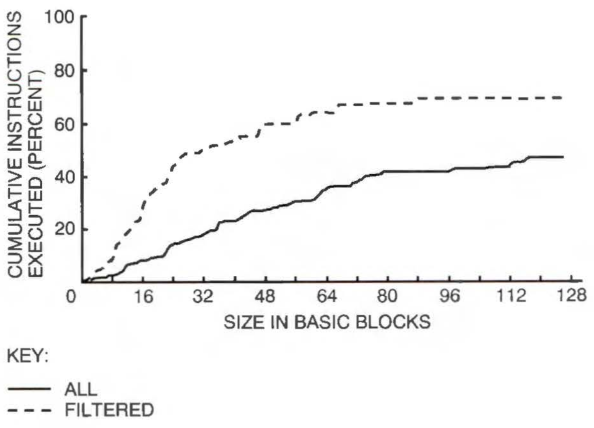


Figure 4
Complexity of Procedures in SQLSERVR for Procedures with an Average Trip Count Less Than 4, Which Account for 69 Percent of the Execution Time

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.

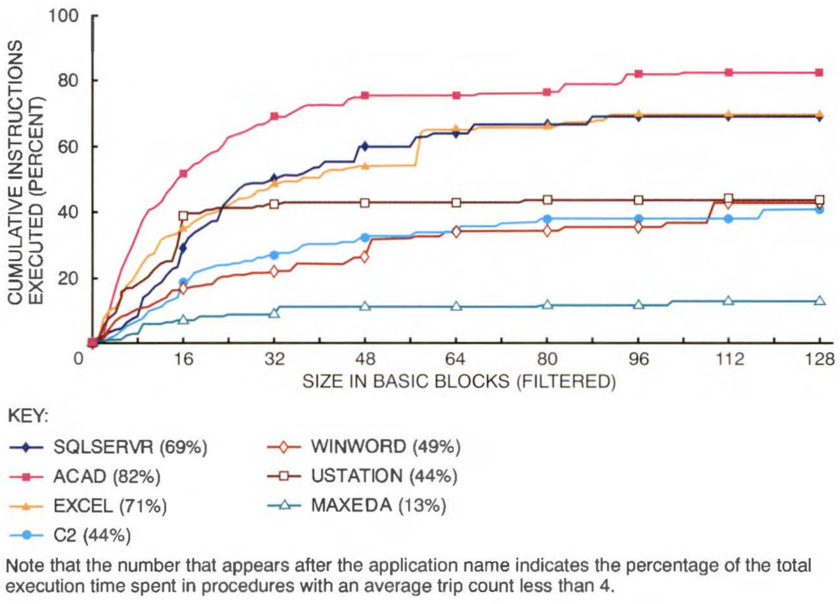
Figure 5 presents the control flow data for many of the Windows NT–based applications listed in Table 1. Again we measured only nonlooping procedures and ignored basic blocks that are rarely executed. Note that all the applications have large procedures. More than half the run time of the nonlooping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

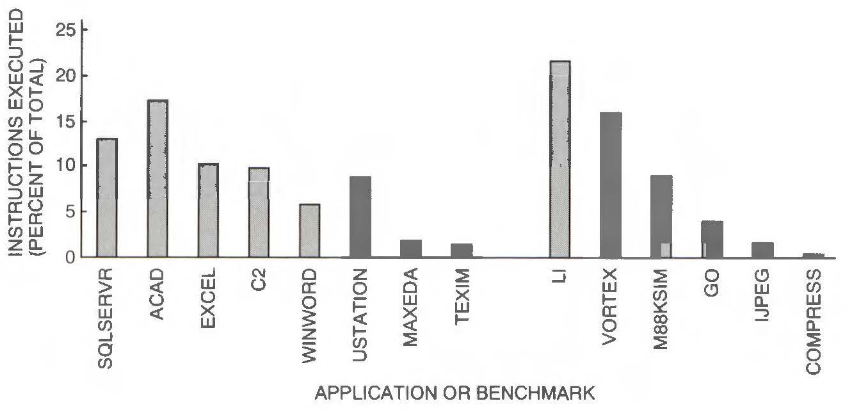


Figure 6
Procedure Call Overhead (Time Spent in Prolog and Epilog)

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-

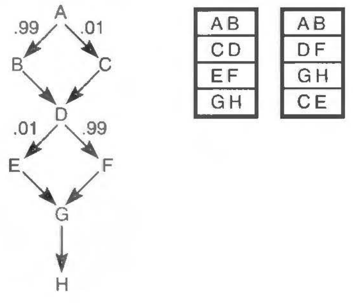


Figure 7
Basic Block Reorganization

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

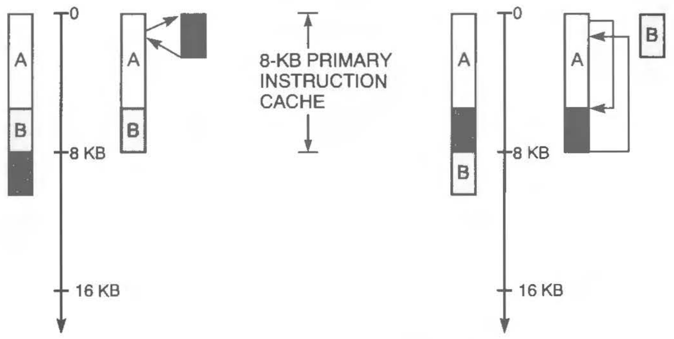


Figure 8
Procedure Placement

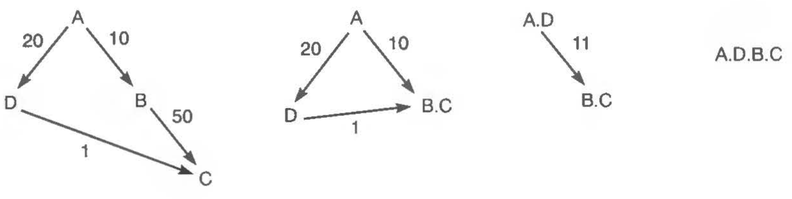
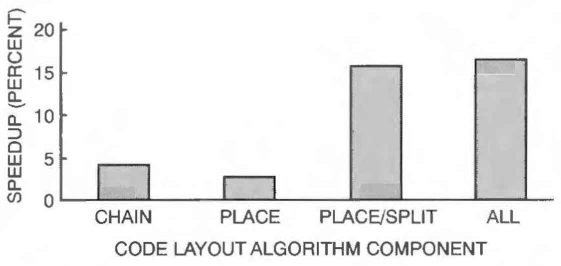


Figure 9
Steps in the Procedure Placement Algorithm

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.

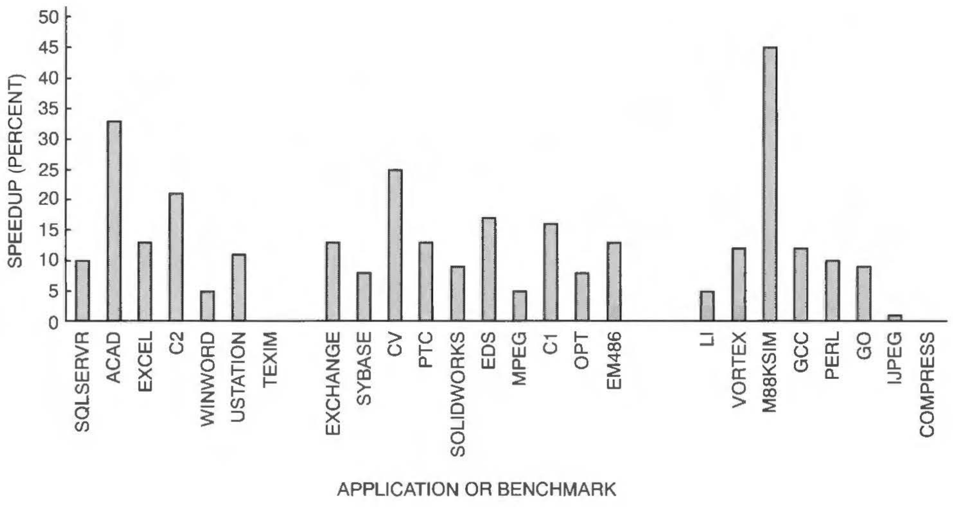


Figure 11
Speedup from Code Layout

```
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.

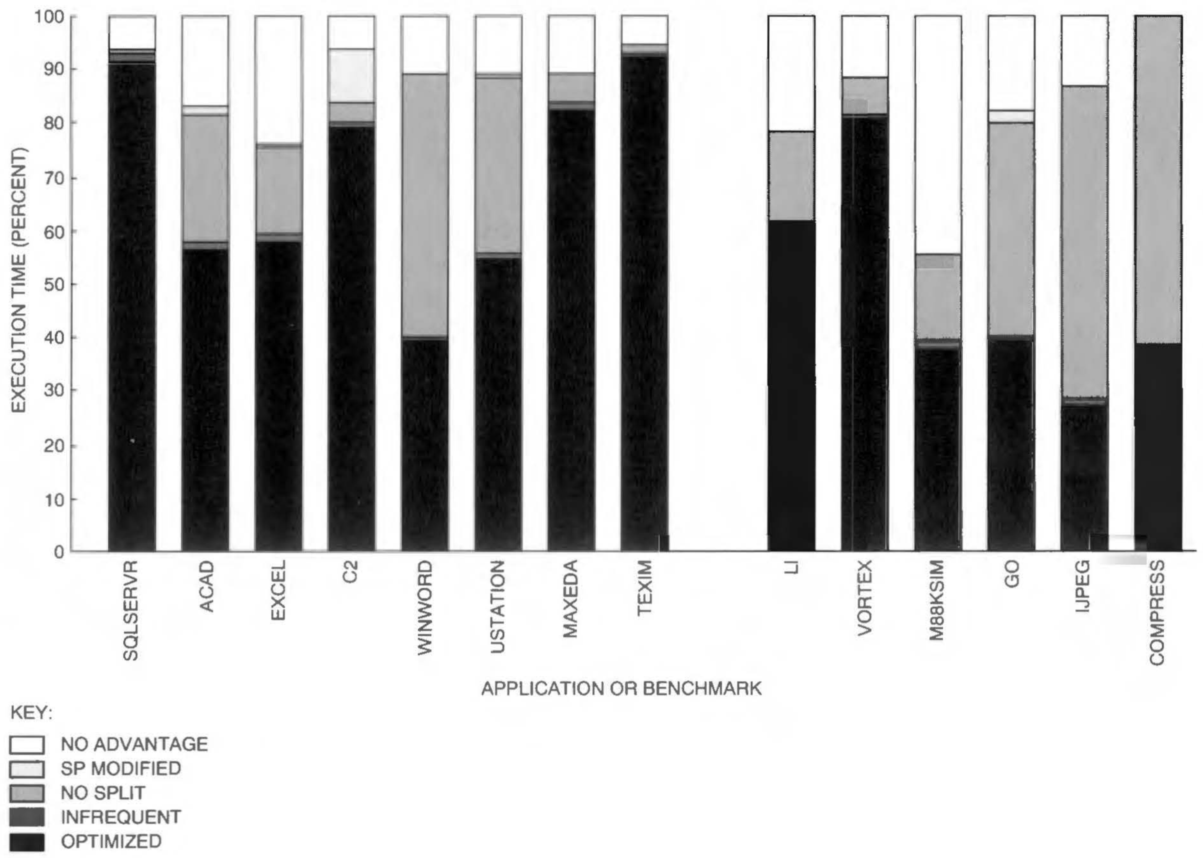


Figure 15
HCO Coverage by Execution Time

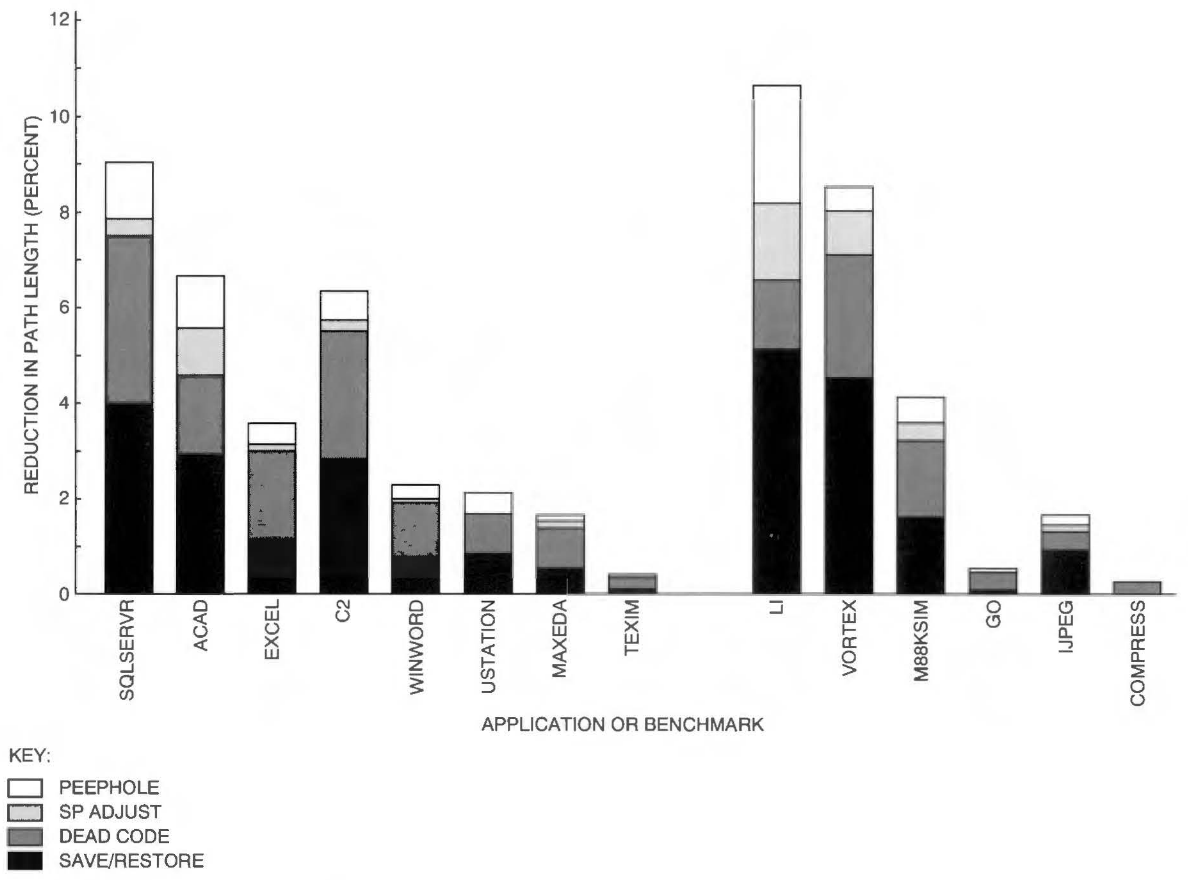


Figure 16
Reduction in Path Length As a Result of HCO

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.

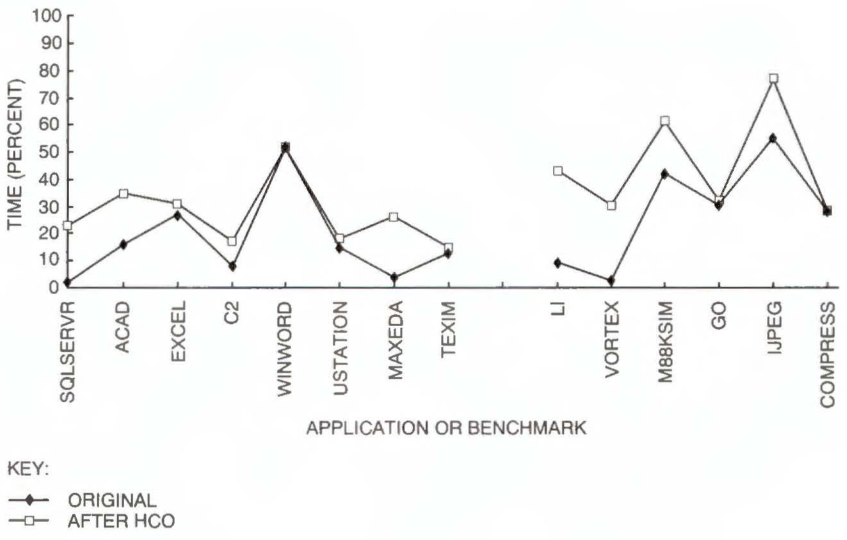


Figure 17
Time Spent in Leaf Procedures before and after HCO

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

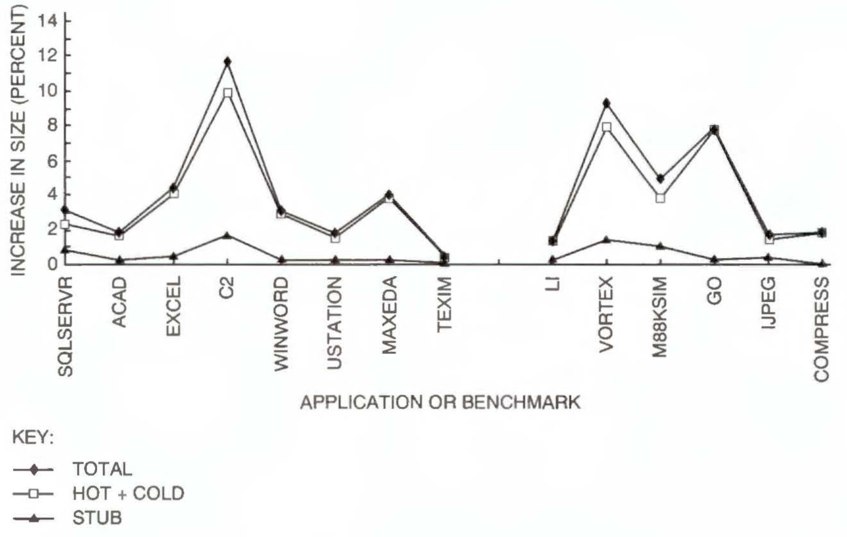


Figure 18
Overall Increase in Code Size after HCO

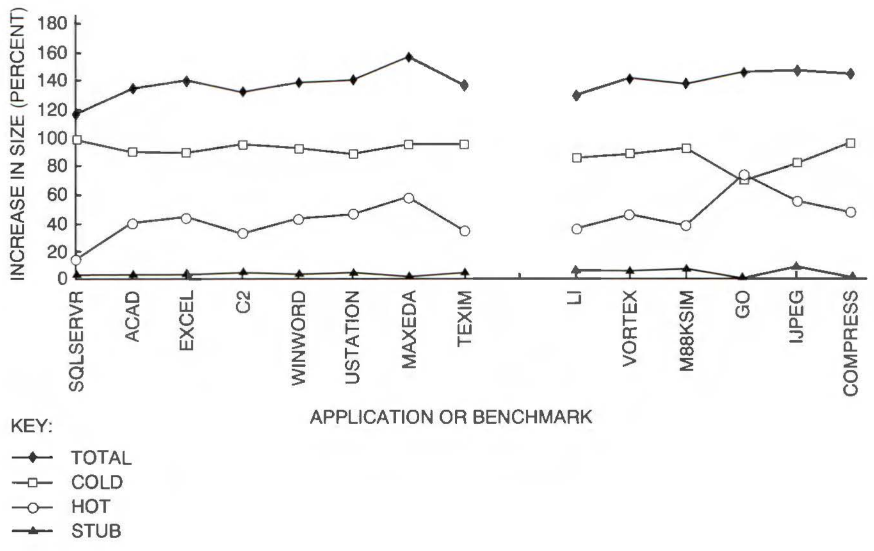


Figure 19
Size of Optimized Procedures after HCO

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.

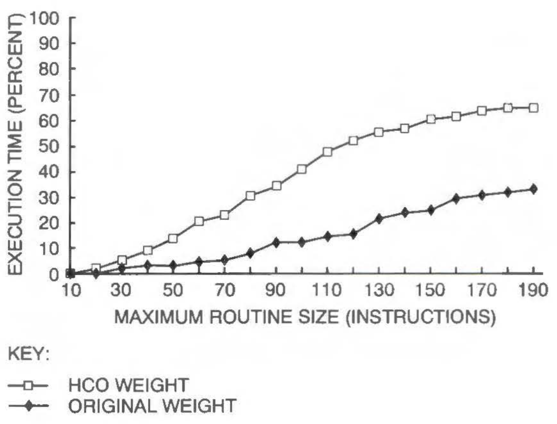
The line labeled TOTAL shows that HCO makes the total code (HOT + COLD + STUB) 20 percent to 60 percent bigger. A procedure is partitioned so that there is less than a 1 percent chance that the stub and cold part are executed, so their size should not have a significant effect on cache behavior as long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

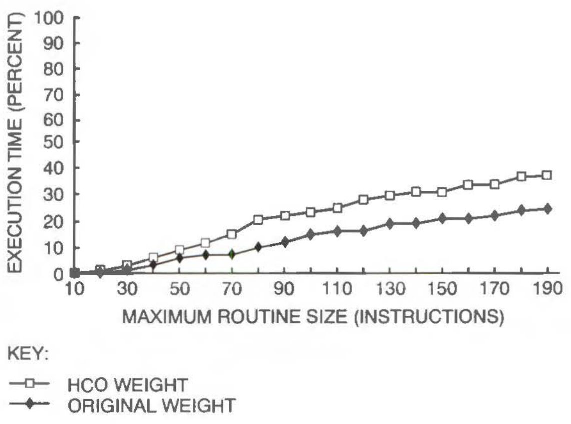
Managing Profile Feedback Optimization

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.

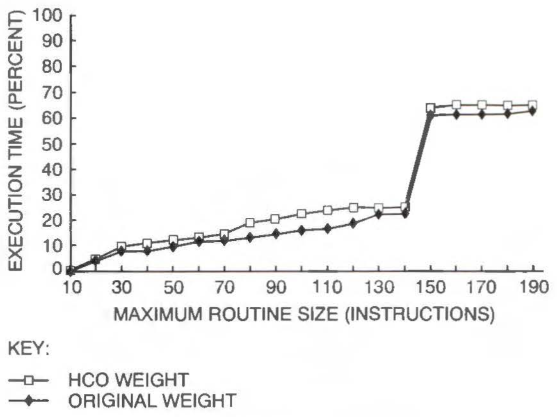
The first step in profile-directed optimization is to instrument each image in an application so that when the application is run, profile information is collected. Instrumentation is most commonly done by using a compiler to insert counters into a program during compilation[18] or by using a post-link 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



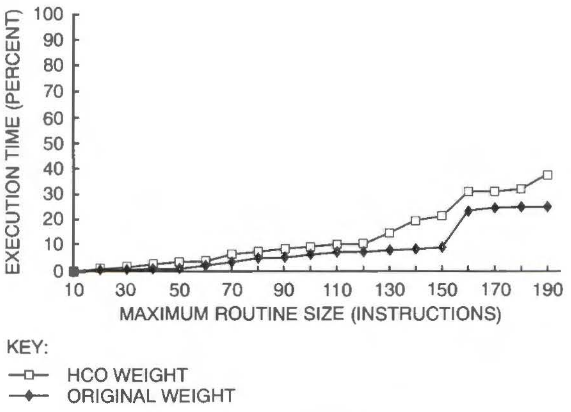
(a) SQLSERVR



(b) C2



(c) WINWORD



(d) MAXEDA

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.

Susanne M. Balle
Simon C. Steely, Jr.

Analyzing Memory Access Patterns of Programs on Alpha-based Architectures

The development of efficient algorithms on today's high-performance computers is far from straightforward. Applications need to take full advantage of the computer system's deep memory hierarchy, and this implies that the user must know exactly how his or her implementation is executed. The ability to understand or predict the execution path without looking at the machine code can be very difficult with today's compilers. By using the outputs from an experimental memory access profiling tool, the programmer can compare memory access patterns of different algorithms and gain insight into the algorithm's behavior, e.g., potential bottlenecks resulting from memory accesses. The use of this tool has helped improve the performance of an application based on sparse matrix-vector multiplications.

The development of efficient algorithms on today's high-performance computers can be a challenge. One major issue in implementing high-performing algorithms is to take full advantage of the deep memory hierarchy. To better understand a program's performance, two things need to be considered: computational intensiveness and the amount of memory traffic involved. In addition to the latter, the pattern of the memory references is important because the success of hierarchy is attributed to locality of reference and reuse of data in the user's program.

In this paper, we investigate the memory access pattern of Fortran programs. We begin by presenting an experimental Atom[1] tool that analyzes how the program is executed. We developed the tool to help us understand how different compiler switches impact the algorithm implemented and to determine if the algorithm is doing what it is intended to do. In addition, our tool helps the process of translating an algorithm into an efficient implementation on a specific machine. The work presented in this paper focuses primarily on a better understanding of the behavior of technical applications. Related work for Basic Linear Algebra Subroutine implementations has been described.[2] In most scientific programs, the data elements are matrix-elements that are usually stored in two-dimensional (2-D) arrays (column-major in Fortran). Knowing the order of array referencing is important in determining the amount of memory traffic.

In the final section of this paper, we present an example of a memory access pattern study and illustrate how the use of our program analysis tool improved the considered algorithm's performance. Guidelines on how to use the tool are given as well as comments about conclusions to be derived from the histograms generated.

Memory Access Profiling Tool

Our experimental tool generates a set of histograms for each reference in the program or in the subroutine under investigation. The first histogram measures strides from the previous reference, the second histogram gives the stride from the second-to-last reference, and so on, for a total of MAXEL histograms for each memory reference in the part of the program we investigate. By stride, we mean the distance between two memory references (load or store). We chose a MAXEL of five for our case study, but MAXEL can be given any value.

Two variants of this tool were implemented.

1. The first version takes all memory references into account in all histograms.
2. The second version takes into account in the next histogram those memory references whose stride is more than 128 bytes. It does not consider in the $(i+1)$th histogram $(i = 1,\ldots,5)$ strides that are less than 128 bytes in the ith histogram.

The second version of the tool has proven to be more useful in understanding the access patterns. It highlights memory accesses that are stride one for a while and then have a stride greater than 128 bytes. The choice of 128 bytes was arbitrary; the value can be changed.

The following bins are used in the histograms: 0- through 127-byte strides are accounted for separately. Strides greater than or equal to 128 bytes are grouped into the following intervals: [128 through 255], [256 through 511], [512 through 1,023], [1,024 through 2,047], [2,048 through 4,095], [4,096 through 8,191], [8,192 through 16,383], [16,384 through 32,767], and [32,768 through infinity].

In the next section, we present the output histograms obtained with the second version of this experimental tool for a Fortran loop. In our case study, we chose to perform the histograms on a single array instead of all references in the program. This method provided a clearer picture of the memory access pattern for each array in the piece of the program under consideration. We present separate histograms for the loads and the stores of each array in the memory traffic of the subroutine we investigated.

When looking at memory access patterns, it is important not to include load instructions that perform prefetching. Even though prefetching adds to the memory traffic, its load instructions pollute the memory access pattern picture.

Case Study

In this section, we study and compare different versions of the code presented in Figure 1 using our experimental memory access profiling tool. We show that the same code is not executed in the same way for different compiler switches. Often a developer has to delve deeply into the assembler of the given loop to understand how and when the different instructions are executed. The output histograms from our tool ease that process and give a clear picture of the reference patterns. The loop presented in Figure 1 implements a sparse matrix-vector multiplication and is part of a larger application. Ninety-six percent of the application's execution time is spent in that loop. We analyze the loop compiled with two different sets of compiler switches. To illustrate the effective use of the tool, we present the enhanced performance results due to changes made based on the output histograms.

```
 1  Q(i)=0, i=1, n
 2  do k1= 1, 4
 3     index = (k1-1) * numrows
 4     do j=1,n
 5         p1=COLSTR(j,k1)
 6         p2=COLSTR(j+1,k1)-1
 7         p3= [snip]
 8         sum0=0.d0
 9         sum1=0.d0
10         sum2=0.d0
11         sum3=0.d0
12         x1 = P(index+ROWIDX(p1,k1))
13         x2 = P(index+ROWIDX(p1+1,k1))
14         x3 = P(index+ROWIDX(p1+2,k1))
15         x4 = P(index+ROWIDX(p1+3,k1))
16         do k = p1, p3, 4
17            sum0 = sum0 + AA(k,k1) * x1
18            sum1 = sum1 + AA(k+1,k1) * x2
19            sum2 = sum2 + AA(k+2,k1) * x3
20            sum3 = sum3 + AA(k+3,k1) * x4
21            x1 = P(index+ROWIDX(k+4,k1))
22            x2 = P(index+ROWIDX(k+5,k1))
23            x3 = P(index+ROWIDX(k+6,k1))
24            x4 = P(index+ROWIDX(k+7,k1))
25         enddo
26         do k = p3+1, p2
27            x1=P(index+ROWIDX(k,k1))
28            sum0 = sum0 + AA(k,k1)*x1
29         enddo
30         YTEMP(j,k1)=sum0+sum1+sum2+sum3
31      enddo
32      do i = 1, n, 4
33         Q(i) = Q(i) + YTEMP(i,k1)
34         Q(i+1) = Q(i+1) + YTEMP(i+1,k1)
35         Q(i+2) = Q(i+2) + YTEMP(i+2,k1)
36         Q(i+3) = Q(i+3) + YTEMP(i+3,k1)
37      enddo
38   enddo

where n = 14000,
real*8 AA(511350,4), YTEMP(n,4)
real*8 Q(n), P(n)
integer*4 ROWIDX(511350,4), COLSTR(n,4)
```

Figure 1
Original Loop

From lines 5 and 6 in the loop shown in Figure 1, we would expect the array *COLSTR* to be read stride one 100 percent of the time. Line 30 of the figure indicates that *YTEMP* is accessed stride one through the whole *j* loop. From lines 33 through 36, we expect *YTEMP*'s stride to be equal to one most of the time and equal to the number of columns in the array every time *k*1 is incremented. *Q* should be referenced 100 percent stride one for both the loads and the stores (lines 33 through 36). As illustrated in lines 12 through 15, 21 through 24, and 27, *ROWIDX* is expected to be accessed with a stride of one between the *p*1 and *p*2 bounds of the *k* loop. Even though it looks like the *k* loop is violating the array bounds of *ROWIDX* in lines 21 through 24 for the last iteration of the loop, this is not the case. We expect array *P* to have nonadjacent memory references since we have deliberately chosen an algorithm that sacrifices this array's access patterns to improve the memory references of *Q* and *AA*.

Original Code

We investigate the memory access patterns achieved by the loop in Figure 1 when compiled with the following switches:

```
f77 -g3 -fast -O5
```

The `-g3` switch is needed to extract the addresses of the arrays from the symbol table. For more information on DIGITAL Fortran compiler switches, see Reference 3.

From Figures 2 and 3, we see that array *Q* is accessed as we expected, 100 percent stride one for the loads and the stores. Since *Q* is accessed contiguously in 100 percent of its memory references, we will not have any entries in the next four histograms. As described in the previous section, we only record in the next histogram the strides that are greater than 128 bytes in the current histogram.

Figure 4 illustrates that *COLSTR* is accessed 50 percent stride zero and 50 percent stride one. This is unexpected since lines 5 and 6 in Figure 1 suggest that this array would be accessed stride one 100 percent of the time. The fact that we have entries only for the strides between the current and the previous loads indicates that the elements of *COLSTR* are accessed in a nearly contiguous way. A closer look at Figure 1 tells us that the compiler is loading *COLSTR* twice. We expected the compiler to do only one load into a register and reuse the register. The work-around is to perform a scalar replacement as described by Blickstein et al.[4] We put $p2 = COLSTR(1,k1) - 1$ outside the *j* loop and substituted inside the *j* loop $p1 = COLSTR(j,k1)$ with $p1 = p2 + 1$. Inside the *j* loop, *p*2 remains the same. Eliminating the extra loads did not enhance performance, and a possible assumption is that the analysis done by the compiler concluded that no gain would result from that optimization.

Figures 5 and 6 show the strides for the loads and the strides for the stores for the array *YTEMP*. One more time, the implementation is not being executed the way we thought it would. In Figure 1, lines 33 through 36 suggested that *YTEMP* would be referenced stride one through the whole *i* loop as well as with a stride

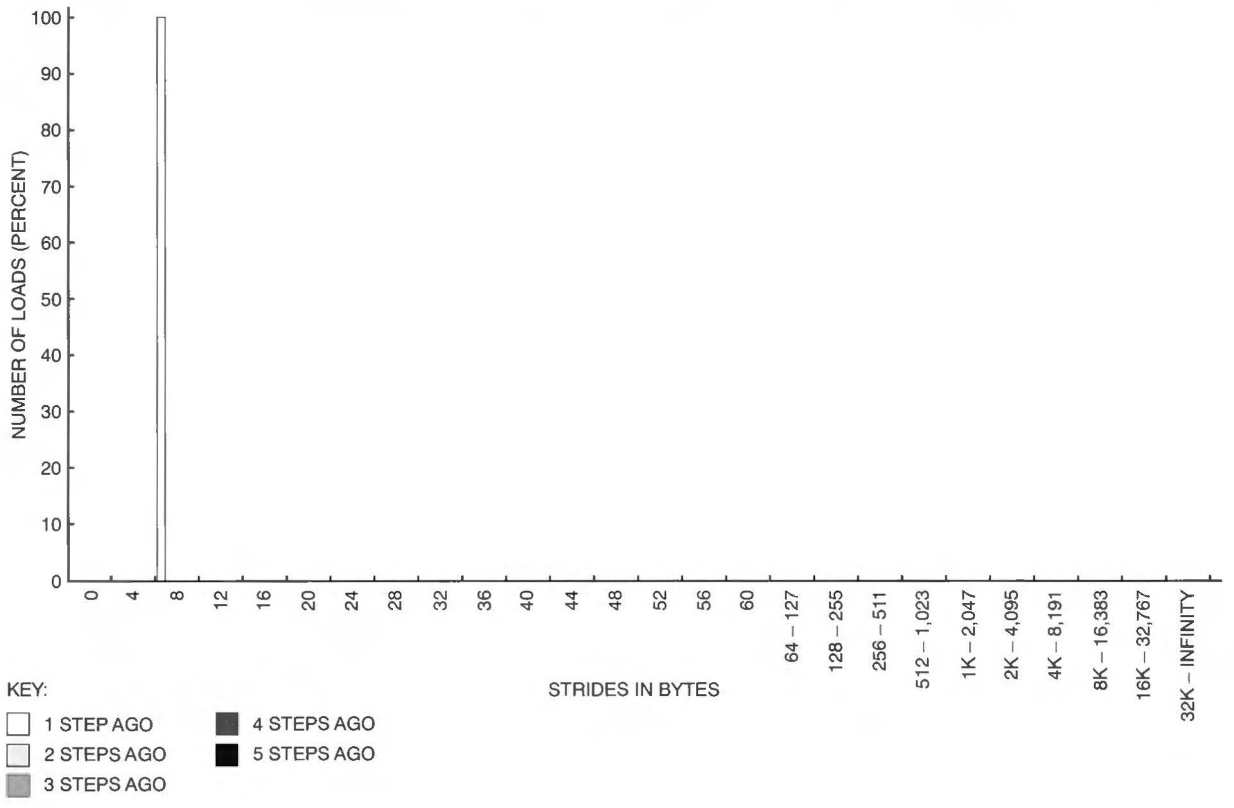


Figure 2
Strides for Array *Q* between the Current Load and the Load One through Five Steps Ago

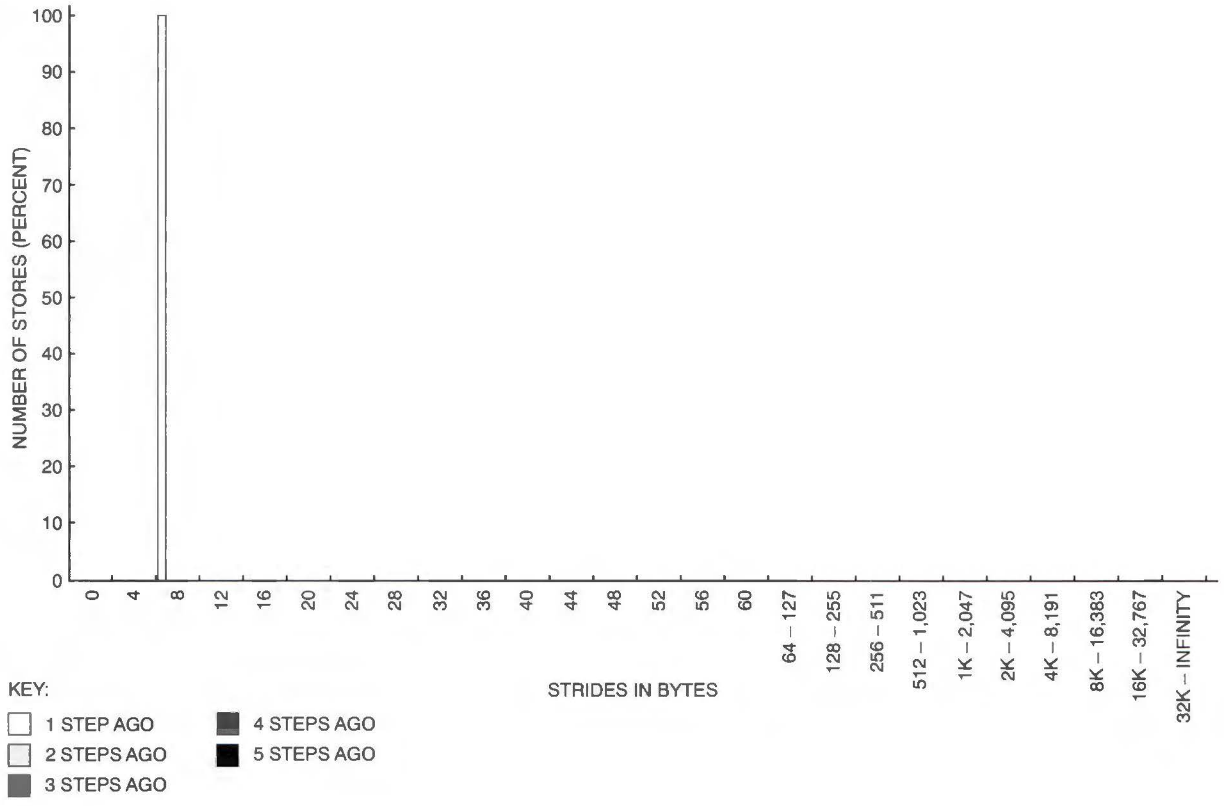


Figure 3
Strides for Array *Q* between the Current Store and the Store One through Five Steps Ago

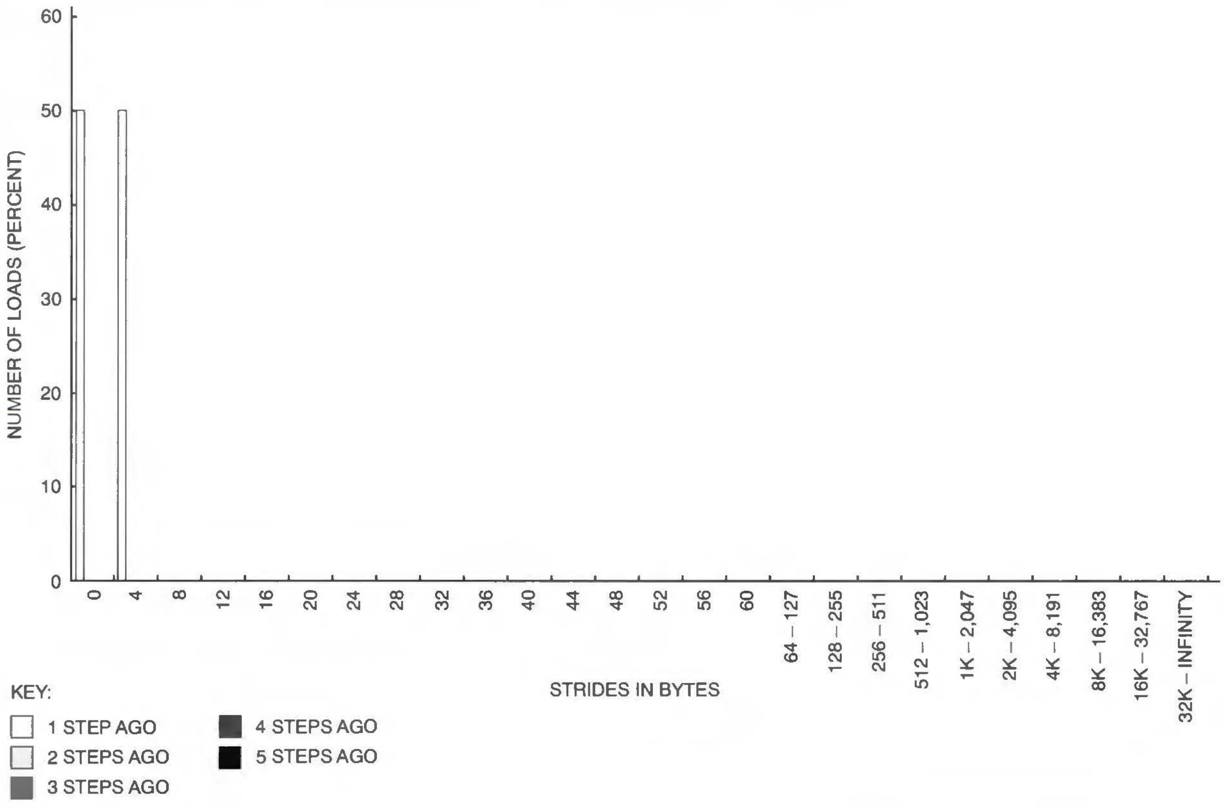


Figure 4
Strides for Array *COLSTR* between the Current Load and the Load One through Five Steps Ago

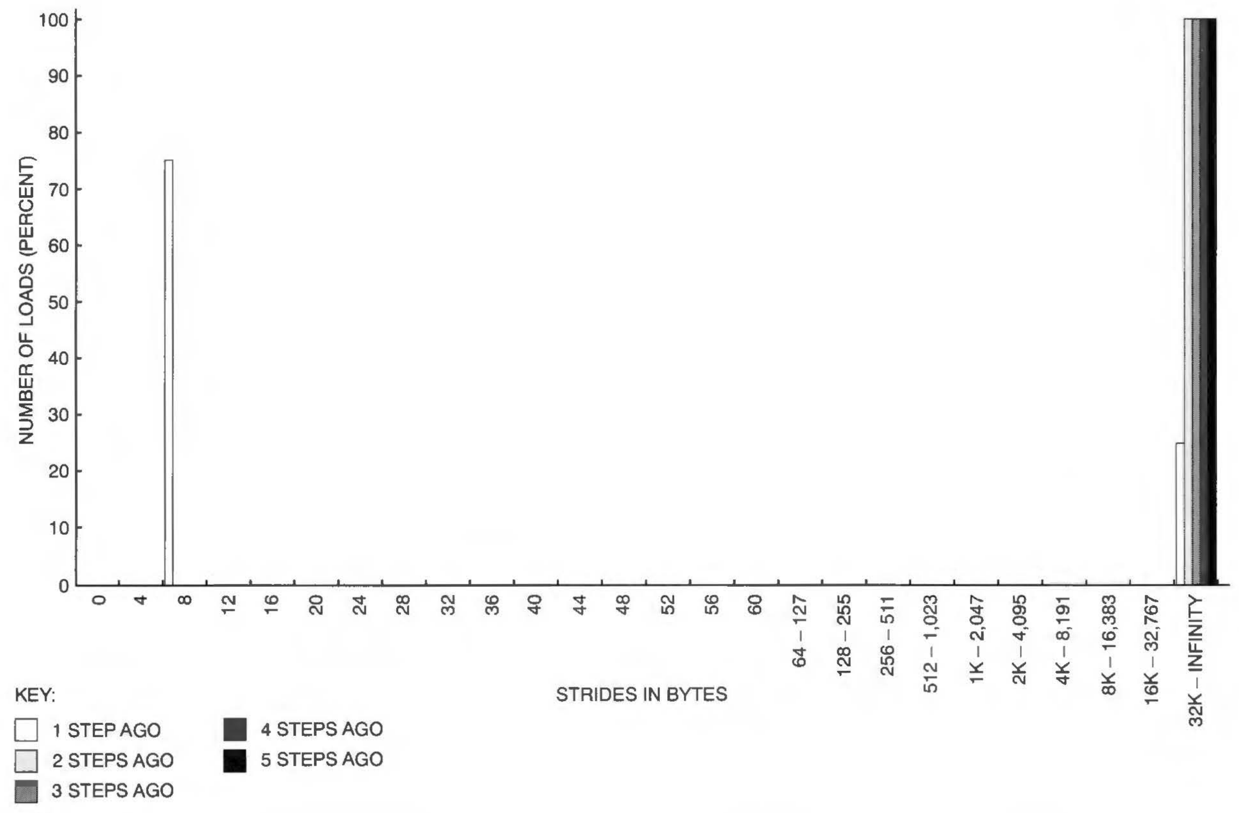


Figure 5
Strides for Array *YTEMP* between the Current Load and the Load One through Five Steps Ago

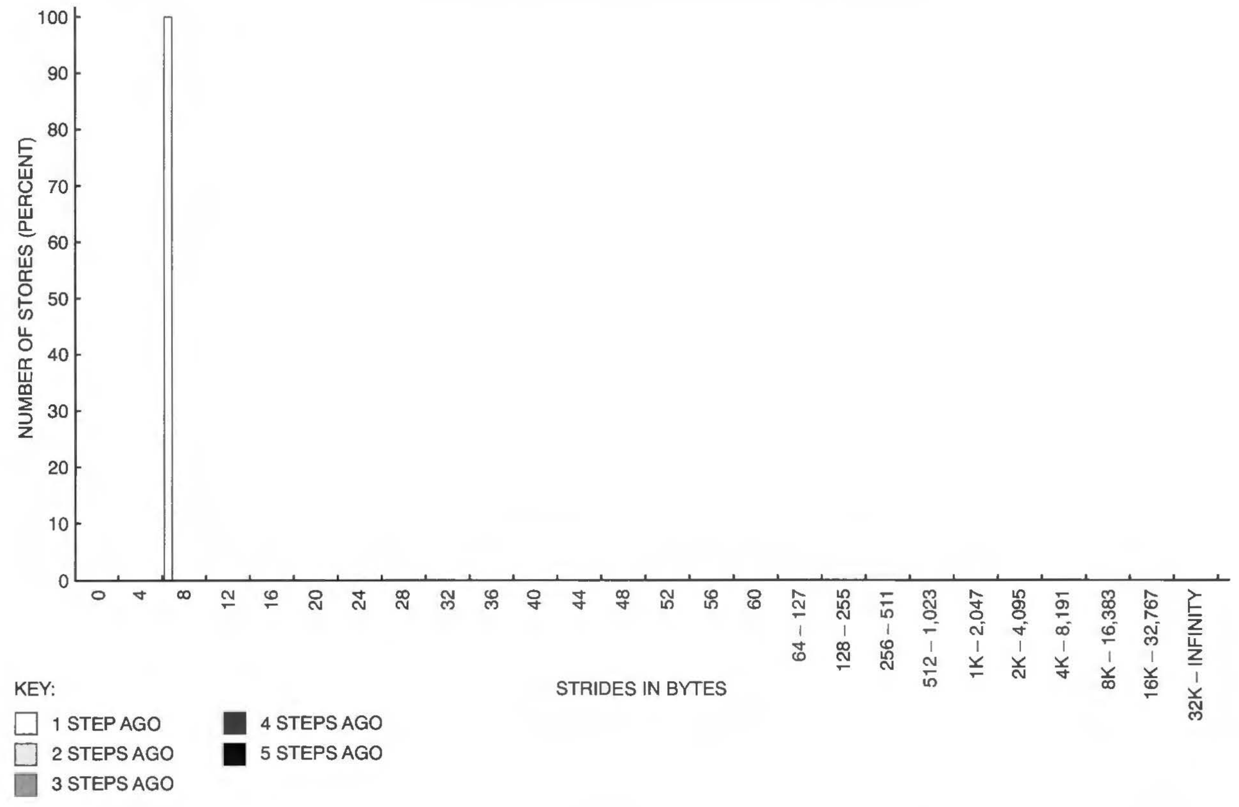


Figure 6
Strides for Array *YTEMP* between the Current Store and the Store One through Five Steps Ago

equal to the number of columns in the array when *k*1 is incremented. By considering Figure 5 along with lines 33 through 36 in Figure 1, we conclude that *YTEMP* is unrolled by four in the *k*1-direction in the *i* loop. The fact that all strides between the current load and the load two loads back or three loads back or four loads back have a stride between 32K and infinity is consistent with traversing a matrix along rows. Figure 6 shows that the *j* loop is not unrolled by four in the *k*1-direction, because all the loads of *YTEMP* are 100 percent stride one. The compiler must split the *k*1 loop into two separate loops, the first consists of the *j* loop and the second consists of the *i* loop. The latter has been unrolled by four in the *k*1-direction thereby eliminating the extra overhead from the *k*1 loop.

Figure 7 shows that the matrix *AA* is accessed as we expected. The strides are not greater than 128 bytes or, in other words, a maximum stride of 16 elements. The fact that there is no stride other than the one between the current load and the previous load in the histograms shows that *AA* is referenced in a controlled way. In this case, *AA* is accessed 39 percent of its total loads in stride one and 23 percent in stride two.

From lines 12 through 15, 17 through 20, and 21 through 24 in Figure 1, we know that the arrays *AA* and *ROWIDX* should have relatively similar behaviors. Only the four extra prefetches of *ROWIDX* in lines 21 through 24 for the last iteration in the *j* loop differentiate the access patterns of the two arrays. Figure 8 confirms that assumption. *ROWIDX* is referenced with controlled strides. Because *ROWIDX* is accessed close to contiguously, we will not have any entries in the next four histograms. As described in the previous section, we only record in the next histogram the strides that are greater than 128 bytes in the current histogram. *ROWIDX* is referenced 24 percent of its total loads in stride one and 34 percent in stride two.

As illustrated in Figure 9, array *P* is accessed exactly the way we expected it. When designing this algorithm, we had to make some compromises. We decided to have *AA* and *Q* referenced as closely as possible to stride one, thus giving up the control of *P*'s references.

By examining these arrays' access patterns, we can see how they are accessed and whether or not the implementation is doing what it is supposed to do. If the loop in Figure 1 is used on a larger matrix [*n* = 75,000 and *AA*(204427,12) has 15 million nonzero elements], the execution time for the total application on a single 21164 processor of an AlphaServer 8400 5/625 system is 1,970 seconds. The application executes 26 x 75 (= 1,950) times the considered loop. When profiling the program, we measured that the loop under investigation takes 96 percent of the total execution time. It is therefore a fair assumption to say that any improvement in this building block will improve the overall performance of the total program.

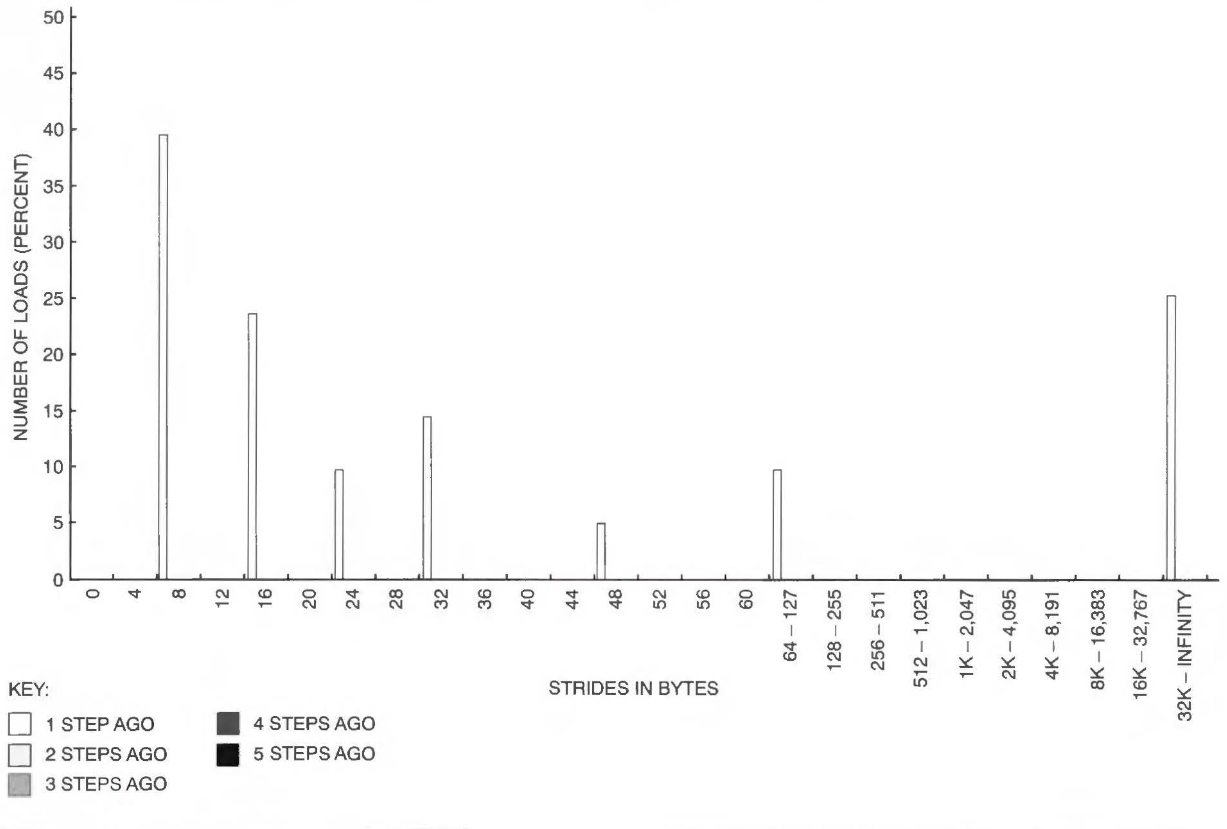


Figure 7
Strides for Array *AA* between the Current Load and the Load One through Five Steps Ago

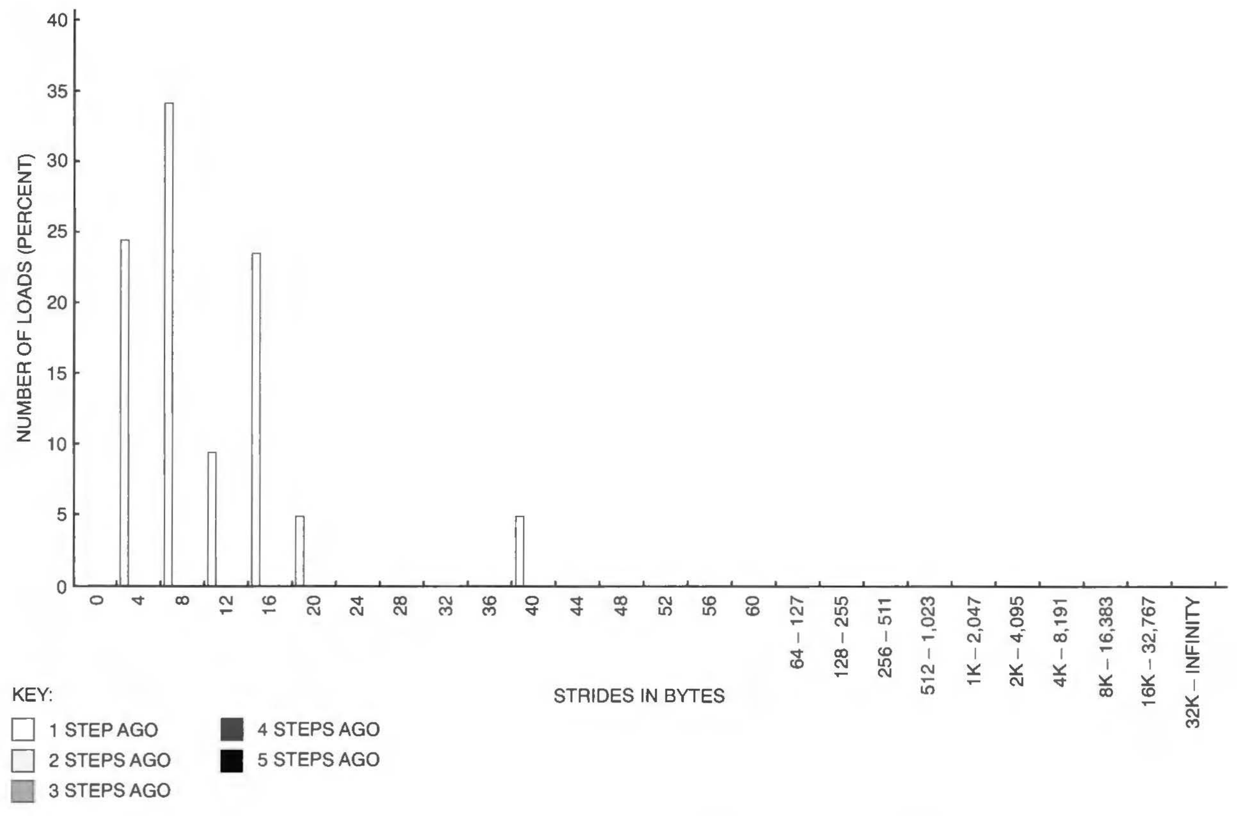


Figure 8
Strides for Array *ROWIDX* between the Current Load and the Load One through Five Steps Ago

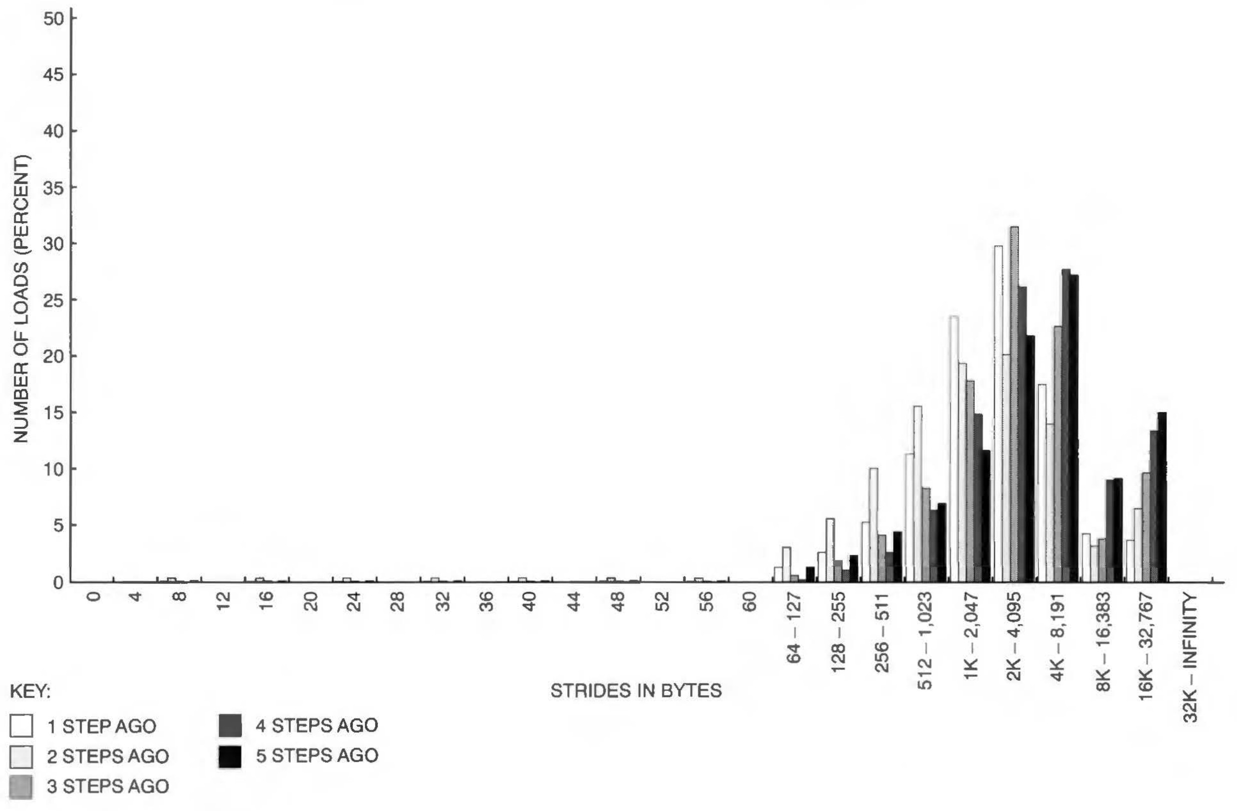


Figure 9
Strides for Array *P* between the Current Load and the Load One through Five Steps Ago

Modified Code

In this section, we describe a new experiment in which we used different compiler switches and changed the original loop to the loop in Figure 10. The code changes were based on the analysis in the previous section as well as on a more extended series of investigations.

In this example, we used the following compiler switches:

```
f77 -g3 -fast -O5 -unroll 1
```

Lines 3, 5, and 6 from Figure 10 show that we implemented the scalar replacement technique as described by Blickstein et al.[4] to avoid *COLSTR* being loaded twice. From Figure 11, we see that array *COLSTR* is now behaving as we expect: 100 percent of the strides for the loads are stride one.

In our first attempt to optimize the original loop, we split the *k*1 loop into two loops in the same way the compiler did as described in the previous section. We then hand unrolled the *YTEMP* array in the *k*1 direction. Further analysis showed that a considerable gain could be made by removing the *YTEMP* array and writing the results directly into *Q*. By replacing the zeroing out of the *Q* array (Figure 1, line 1) with an IF statement (Figure 10, line 30), we further improved the performance of the loop. The last two changes were possible because we decided that, for performance enhancement issues, the serial version of the code was going to be different from its parallel version.

Figures 12 and 13 show that *Q*'s load and store access pattern is 100 percent stride one as we expected it to be. For both *ROWIDX* and *AA*, we see a significant increase in stride one references. Figure 14 shows that *AA* is now accessed 69 percent stride one instead of 39 percent. *ROWIDX*'s stride one increased to 52 percent from 24 percent as illustrated in Figure 15. These two arrays are the reason for using the `-unroll 1` switch. Without it, stride one for both arrays would stay approximately the same as in the previous study. The pattern of accesses of array *P* in Figure 16 is similar to the prior pattern of accesses in Figure 9 as expected.

To better understand the effects of the unrolling, we counted the number of second-level cache misses for 26 calls to the loop, using an Atom tool[1] that simulated a 4-megabyte direct-mapped cache. By considering only these 26 matrix-vector multiplications, we do not get a full picture of what is going on and how the different arrays interact. Nevertheless, it gives us hints about what caused the improvement in performance. Use of the cache tool on the whole application would increase the run time dramatically.

Twenty-six calls to the original loop (Figure 1) have a total of 1,476,017,322 memory references, of which 77,638,624 are cache misses. The modified loop (Figure 10), on the other hand, has fewer references due to the fact that we eliminated an expensive array initialization at each step and removed the temporary array *YTEMP*. The number of cache misses dropped from 77,638,624 to 72,384,348 or a reduction in misses of 7 percent. If we compile the modified loop without the `-unroll 1` switch, the number of cache misses increases slightly. On the 21164 Alpha microprocessor, all the misses are effectively performed in serial. This means that for memory-bound codes like the loop we are currently investigating, execution time primarily depends on the number of cache misses.

The histograms illustrating the access strides for the different arrays helped us design a more suitable algorithm for our architecture. By increasing the stride one references in the loads for the arrays *AA* and *ROWIDX*, eliminating the extra references in *COLSTR* and *Q*, and improving the strides for *Q*, we increased the performance of this application dramatically. Counting the number of cache misses gave us a better understanding as to why the new access patterns achieve enhanced performance. It also helped us understand that not allowing the compiler to unroll the already hand-unrolled loops in the modified loop decreased the number of cache misses. The execution time for this application [n = 75,000 and *AA*(204427,12) has 15 million nonzero elements] decreased from 1,970 seconds to 1,831 seconds on a single 625-megahertz (MHz) 21164 Alpha microprocessor of an AlphaServer 8400 5/625 system. This is an improvement of 139 seconds or 8 percent.

```
1  do k1= 1, 4
2     index = (k1-1) * numrows
3     p2=COLSTR(1,k1)-1
4     do j=1,n
5        p1=p2+1
6        p2=COLSTR(j+1,k1)-1
7        p3= [snip]
8        sum0=0.d0
9        sum1=0.d0
10       sum2=0.d0
11       sum3=0.d0
12       x1 = P(index+ROWIDX(p1,k1))
13       x2 = P(index+ROWIDX(p1+1,k1))
14       x3 = P(index+ROWIDX(p1+2,k1))
15       x4 = P(index+ROWIDX(p1+3,k1))
16       do k = p1, p3, 4
17          sum0 = sum0 + AA(k,k1) * x1
18          sum1 = sum1 + AA(k+1,k1) * x2
19          sum2 = sum2 + AA(k+2,k1) * x3
20          sum3 = sum3 + AA(k+3,k1) * x4
21          x1 = P(index+ROWIDX(k+4,k1))
22          x2 = P(index+ROWIDX(k+5,k1))
23          x3 = P(index+ROWIDX(k+6,k1))
24          x4 = P(index+ROWIDX(k+7,k1))
25       enddo
26       do k = p3+1, p2
27          x1=P(index+ROWIDX(k,k1))
28          sum0 = -sum0 + AA(k,k1)*x1
29       enddo
30       if(k1.eq.1) then
31          Q(j) = sum0 + sum1 + sum2 + sum3
32       else
33          Q(j) = Q(j) + sum0 + sum1 + sum2 + sum3
34       endif
35    enddo
36  enddo

where n = 14000,
real*8 AA(511350,4)
real*8 Q(n), P(n)
integer*4 ROWIDX(511350,4), COLSTR(n,4)
```

Figure 10
Modified Loop

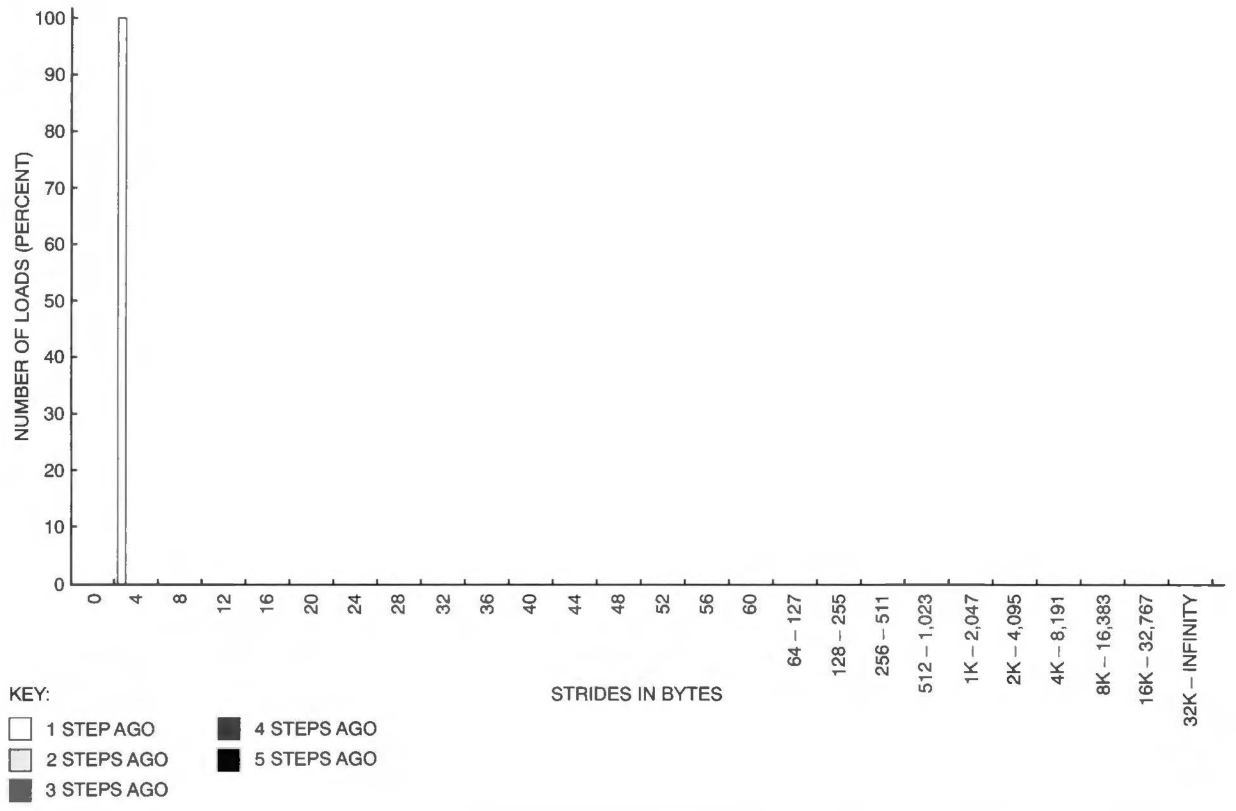


Figure 11
Strides for Array *COLSTR* between the Current Load and the Load One through Five Steps Ago

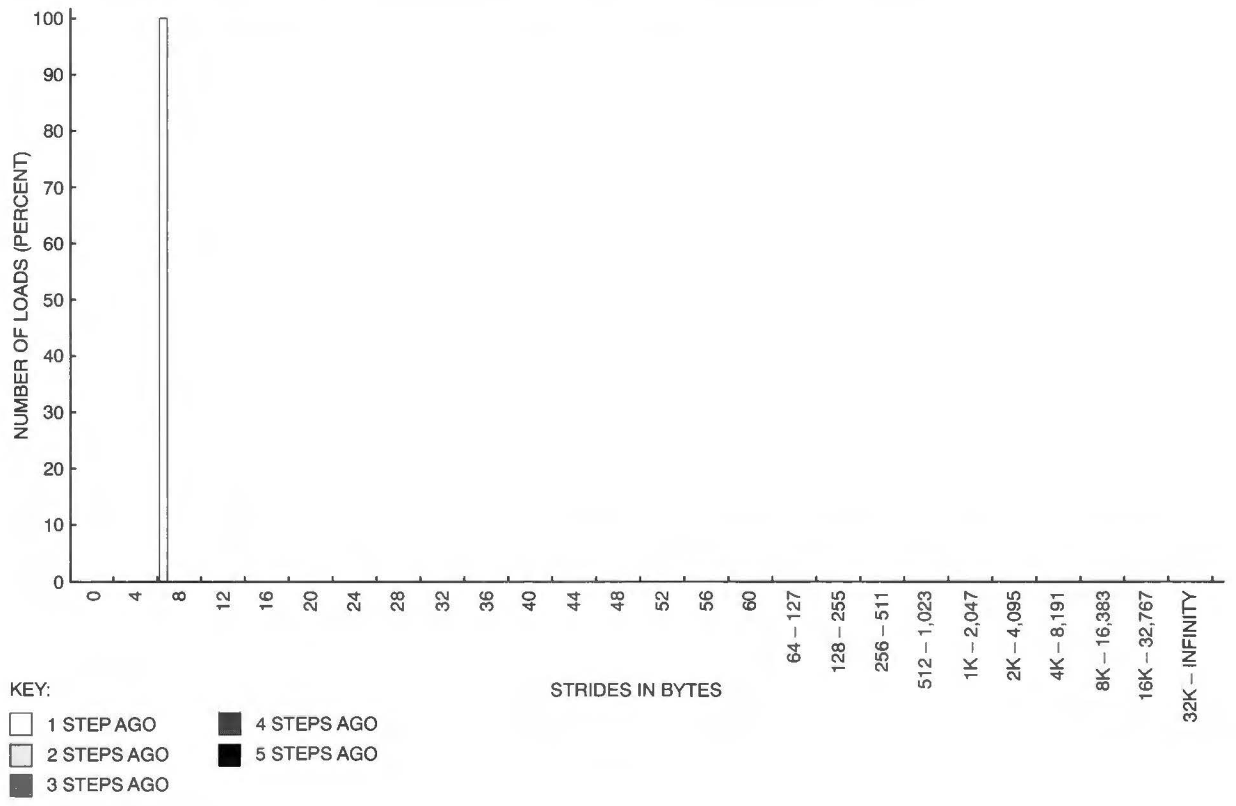


Figure 12
Strides for Array *Q* between the Current Load and the Load One through Five Steps Ago

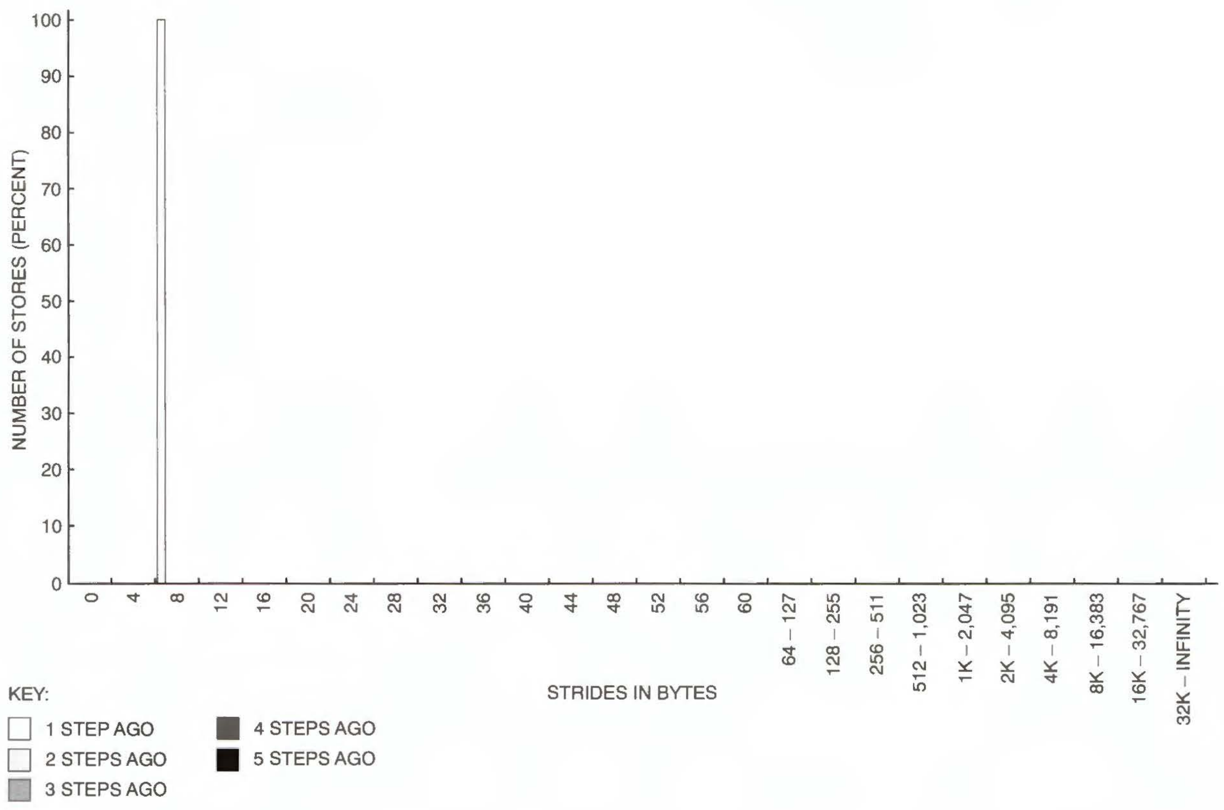


Figure 13
Strides for Array *Q* between the Current Store and the Store One through Five Steps Ago

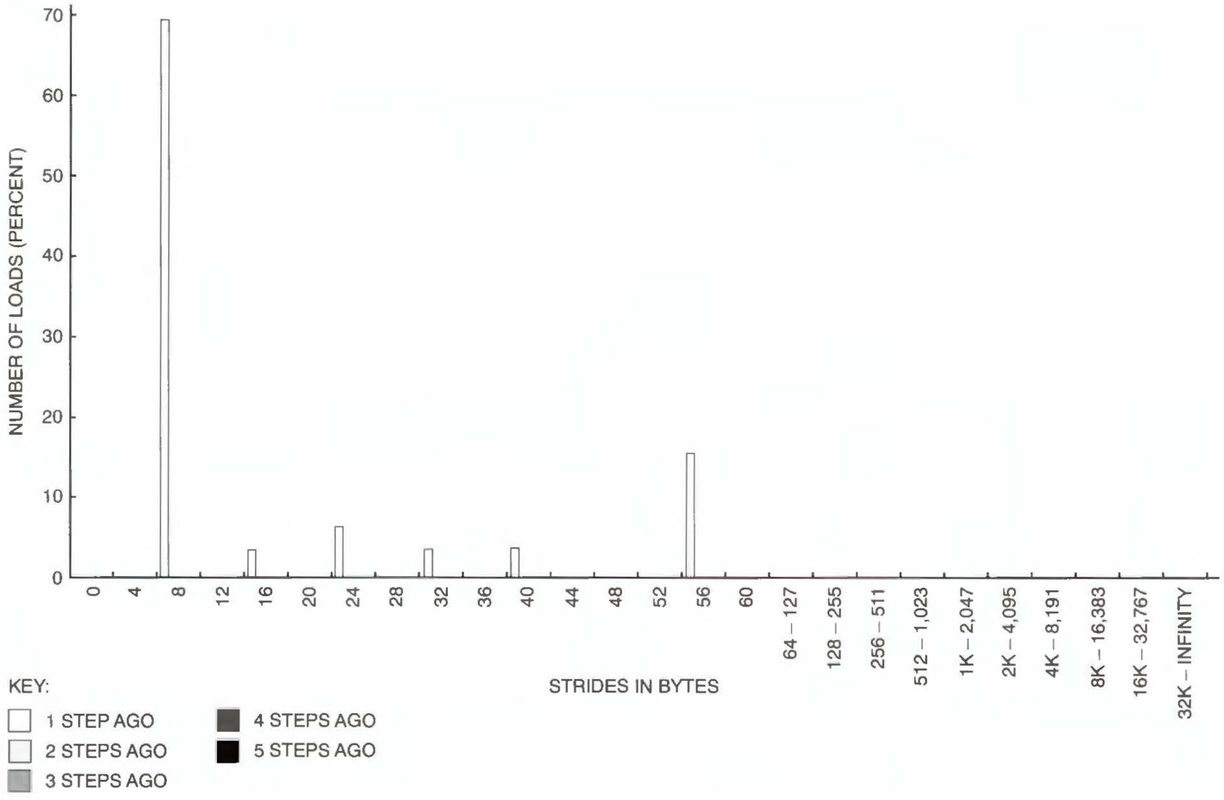


Figure 14
Strides for Array *AA* between the Current Load and the Load One through Five Steps Ago

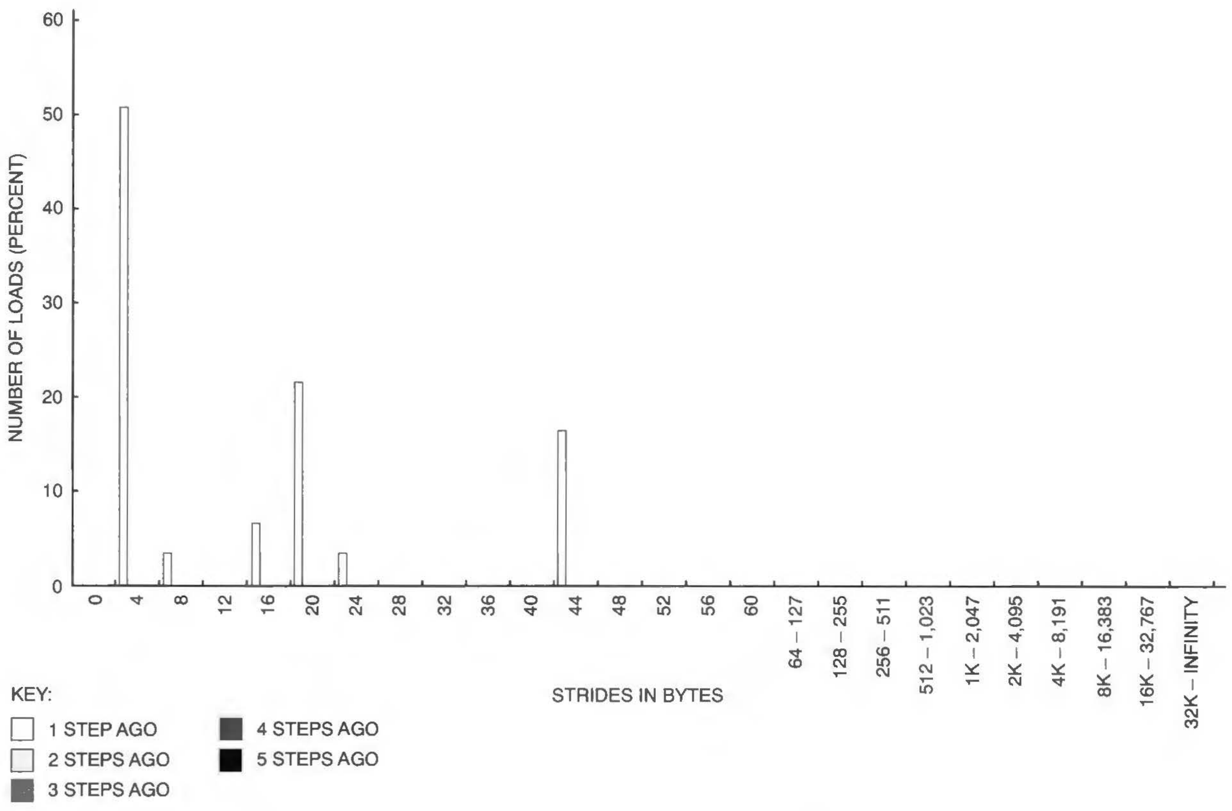


Figure 15
Strides for Array *ROWIDX* between the Current Load and the Load One through Five Steps Ago

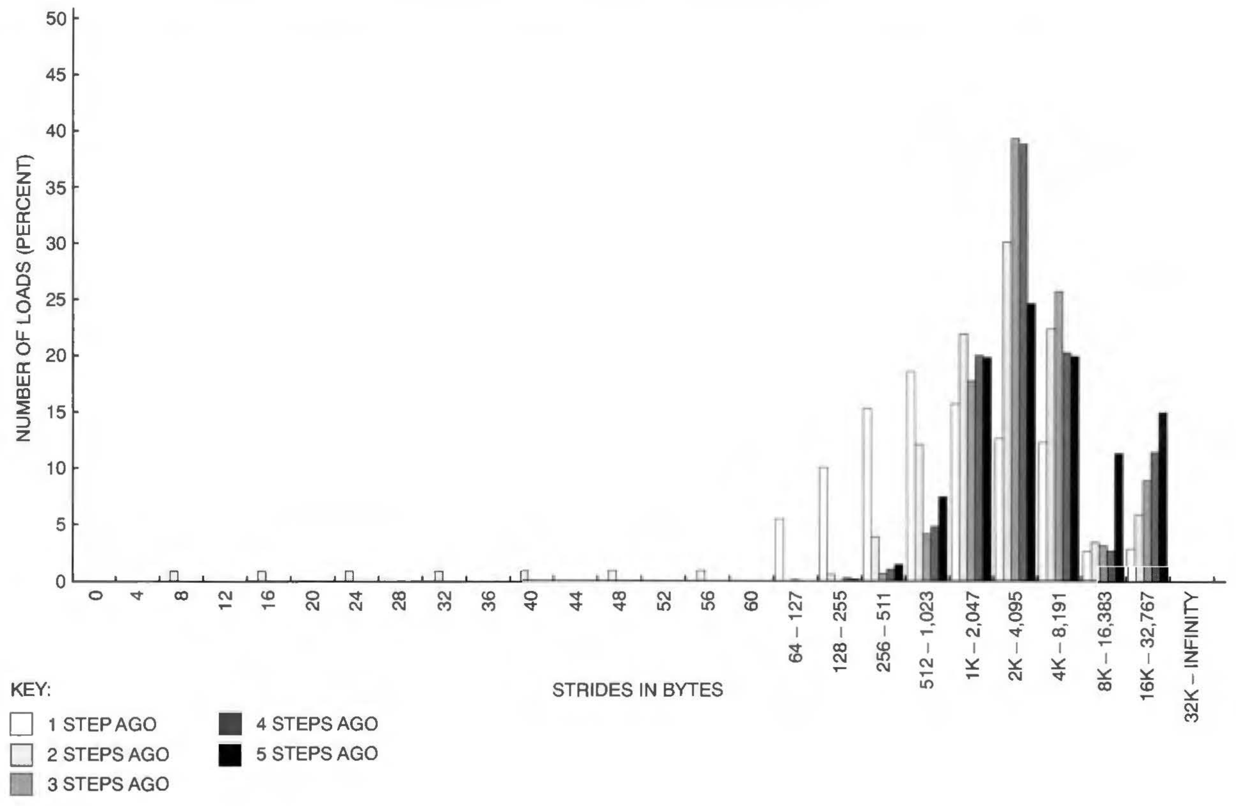


Figure 16
Strides for Array *P* between the Current Load and the Load One through Five Steps Ago

Conclusion

The case study shows that, given the right program analysis tools, a program developer can take better advantage of his or her computer system. The experimental tool we designed was very useful in providing insight into the algorithm's behavior. The approach considered yields an improvement in performance of 8 percent on a 625-MHz 21164 Alpha microprocessor. This is definitely a worthwhile exercise since a substantial reduction in execution time was obtained using straightforward and easy guidelines.

The data collected from a memory access profiling tool helps the user understand a given program as well as its memory access patterns. It is an easier and faster way to gain insight into a program than examining the listing and the assembler generated by the compiler. Such a tool enables the programmer to compare memory access patterns of different algorithms; therefore, it is very useful when optimizing codes. Probably its most important value is that it shows the developer if his or her implementation is doing what he or she thinks the algorithm is doing and highlights potential bottlenecks resulting from memory accesses. Optimizing an application is an iterative process, and being able to use relatively easy-to-use tools like Atom is a very important part of the process. The major advantage of the tool presented in this paper is that no source code is needed, so it can be used to analyze the performance of program executables.

Acknowledgments

The authors wish to thank David LaFrance-Linden, Dwight Manley, Jean-Marie Verdun, and Dick Foster for fruitful discussions. Thanks to John Eck, Dwight Manley, Ned Anderson, and Chuck Schneider for reading and commenting on earlier versions of the paper. Special thanks go to the anonymous referees for comments and suggestions that helped us considerably improve the paper. Finally, we thank John Kreatsoulas and Dave Fenwick for encouraging our collaboration.

References

1. *Programmer's Guide, Digital UNIX Version 4.0*, chapter 9 (Maynard, Mass., Digital Equipment Corporation, March 1996).

2. O. Brewer, J. Dongarra, and D. Sorensen, *Tools to Aid in the Analysis of Memory Access Patterns for Fortran Programs*, Technical Report, Argonne National Laboratory (June 1988).

3. *DEC Fortran Language Reference Manual* (Maynard, Mass., Digital Equipment Corporation, 1997).

4. D. Blickstein, P. Craig, C. Davidson, R. Faiman, K. Glossop, R. Grove, S. Hobbs, and W. Noyce, The GEM Optimizing Compiler System, *Digital Technical Journal*, vol. 4, no. 4 (1992): 121–136.

Biographies

Susanne M. Balle
Susanne Balle is currently a member of the High Performance Computing Expertise Center. Her work areas are performance prediction for 21264 microprocessor-based architectures, memory access pattern analysis, parallel Lanczos algorithms for solving very large eigenvalue problems, distributed-memory matrix computations, as well as improving performance of standard industry and customer benchmarks. Before joining DIGITAL, she was a postdoctoral fellow at the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center where she worked on high-performance mathematical software. From 1992 to 1995, she consulted for the Connection Machine Scientific Software Library (CMSSL) group at Thinking Machines Corporation. Susanne received a Ph.D. in mathematics and an M.S. in mechanical engineering and computational fluid dynamics from the Technical University of Denmark. She is a member of SIAM.

Simon C. Steely, Jr.
Simon Steely is a consulting engineer in DIGITAL's AlphaServer Platform Development Group. In his 21 years at DIGITAL, he has worked on many projects, including development of the PDP-11, VAX, and Alpha systems. His work has focused on writing microcode, designing processor and system architecture, and doing performance analysis to make design decisions. In his most recent project, he was a member of the architecture team for a future system. In addition, he led the team developing the cache-coherency protocol on that project. His primary interests are computer architecture, performance analysis, prediction technologies, cache/memory hierarchies, and optimization of code for best performance. Simon has a B.S. in engineering from the University of New Mexico and is a member of IEEE and ACM. He holds 15 patents and has several more pending.

Karen L. Noel
Nitin Y. Karkhanis

OpenVMS Alpha 64-bit Very Large Memory Design

The OpenVMS Alpha version 7.1 operating system provides memory management features that extend the 64-bit VLM capabilities introduced in version 7.0. The new OpenVMS Alpha APIs and mechanisms allow 64-bit VLM applications to map and access very large shared memory objects (global sections). Design areas include shared memory objects without disk file backing storage (memory-resident global sections), shared page tables, and a new physical memory and system fluid page reservation system.

Database products and other applications impose heavy demands on physical memory. The newest version of DIGITAL's OpenVMS Alpha operating system extends its very large memory (VLM) support and allows large caches to remain memory resident. OpenVMS Alpha version 7.1 enables applications to take advantage of both 64-bit virtual addressing and very large memories consistent with the OpenVMS shared memory model. In this paper, we describe the new 64-bit VLM capabilities designed for the OpenVMS Alpha version 7.1 operating system. We explain application flexibility and the system management issues addressed in the design and discuss the performance improvements realized by 64-bit VLM applications.

Overview

A VLM system is a computer with more than 4 gigabytes (GB) of main memory. A flat, 64-bit address space is commonly used by VLM applications to address more than 4 GB of data.

A VLM system allows large amounts of data to remain resident in main memory, thereby reducing the time required to access that data. For example, database cache designers implement large-scale caches on VLM systems in an effort to improve the access times for database records. Similarly, VLM database applications support more server processes than ever before. The combination of large, in-memory caches and an increased number of server processes significantly reduces the overall time database clients wait to receive the data requested.[1]

The OpenVMS Alpha version 7.0 operating system took the first steps in accommodating the virtual address space requirements of VLM applications by introducing 64-bit virtual addressing support. Prior to version 7.0, large applications—as well as the OpenVMS operating system itself—were becoming constrained by the limits imposed by a 32-bit address space.

Although version 7.0 eased address space restrictions, the existing OpenVMS physical memory management model did not scale well enough to accommodate VLM systems. OpenVMS imposes specific limits on the amount of physical memory a process can occupy. As a result, applications lacked the ability to keep a very large object in physical memory. In systems on which the physical memory is not plentiful, the mechanisms that limit per-process memory utilization serve to ensure fair-and-equal access to a potentially scarce resource. However, on systems rich with memory whose intent is to service applications creating VLM objects, the limitations placed on per-process memory utilization inhibit the overall performance of those applications. As a result, the benefits of a VLM system may not be completely realized.

Applications that require very large amounts of physical memory need additional VLM support. The goals of the OpenVMS Alpha VLM project were the following:

- Maximize the operating system's 64-bit capabilities
- Take full advantage of the Alpha Architecture
- Not require excessive application change
- Simplify the system management of a VLM system
- Allow for the creation of VLM objects that exhibit the same basic characteristics, from the programmer's perspective, as other virtual memory objects created with the OpenVMS system service programming interface

These goals became the foundation for the following VLM technology implemented in the OpenVMS Alpha version 7.1 operating system:

- Memory-resident global sections—shared memory objects that do not page to disk
- Shared page tables—page tables mapped by multiple processes, which in turn map to memory-resident global sections
- The reserved memory registry—a memory reservation system that supports memory-resident global sections and shared page tables

The remainder of this paper describes the major design areas of VLM support for OpenVMS and discusses the problems addressed by the design team, the alternatives considered, and the benefits of the extended VLM support in OpenVMS Alpha version 7.1.

Memory-resident Global Sections

We designed memory-resident global sections to resolve the scaling problems experienced by VLM applications on OpenVMS. We focused our design on the existing shared memory model, using the 64-bit addressing support. Our project goals included simplifying system management and harnessing the speed of the Alpha microprocessor. Before describing memory-resident global sections, we provide a brief explanation of shared memory, process working sets, and a page fault handler.

Global Sections

An OpenVMS global section is a shared memory object. The memory within the global section is shared among different processes in the system. Once a process has created a global section, others may map to the section to share the data. Several types of global sections can be created and mapped by calling OpenVMS system services.

Global Section Data Structures Internally, a global section consists of several basic data structures that are stored in system address space and are accessible to all processes from kernel mode. When a global section is created, OpenVMS allocates and initializes a set of these data structures. The relationship between the structures is illustrated in Figure 1. The sample global section is named "SHROBJ" and is 2,048 Alpha pages or 16 megabytes (MB) in size. Two processes have mapped to the global section by referring to the global

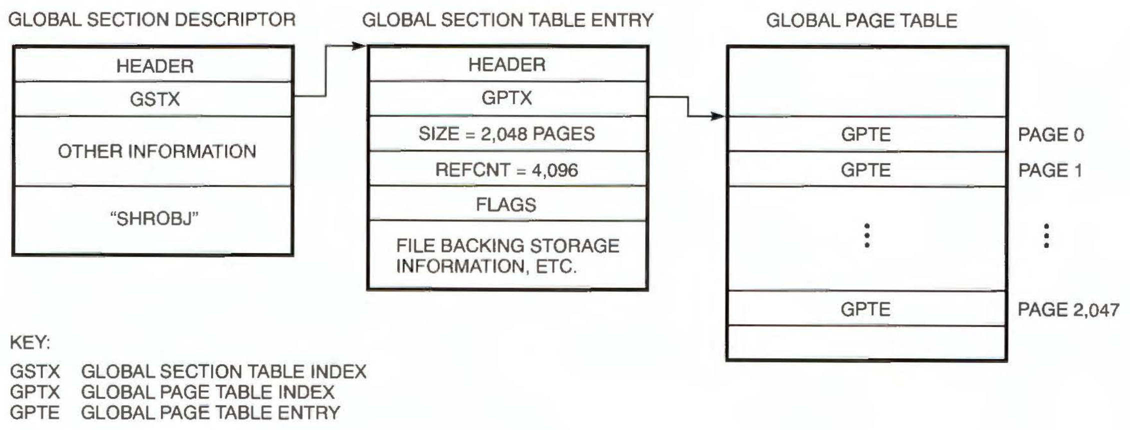

GLOBAL SECTION DESCRIPTOR
HEADER
GSTX
OTHER INFORMATION
"SHROBJ"
GLOBAL SECTION TABLE ENTRY
HEADER
GPTX
SIZE = 2,048 PAGES
REFCNT = 4,096
FLAGS
FILE BACKING STORAGE INFORMATION, ETC.
GLOBAL PAGE TABLE
GPTE
PAGE 0
GPTE
PAGE 1
GPTE
PAGE 2,047
KEY:
GSTX GLOBAL SECTION TABLE INDEX
GPTX GLOBAL PAGE TABLE INDEX
GPTE GLOBAL PAGE TABLE ENTRY

Figure 1
Global Section Data Structures

section data structures in their process page table entries (PTEs).

Process PTEs Mapping to Global Sections When a process maps to a global section, its process PTEs refer to global section pages in a one-to-one fashion. A page of physical memory is allocated when a process accesses a global section page for the first time. This results in both the process PTE and the global section page becoming valid. The page frame number (PFN) of the physical page allocated is stored in the process PTE. Figure 2 illustrates two processes that have mapped to the global section where the first process has accessed the first page of the global section.

When the second process accesses the same page as the first process, the same global section page is read from the global section data structures and stored in the process PTE of the second process. Thus the two processes map to the same physical page of memory.

The operating system supports two types of global sections: a global section whose original contents are zero or a global section whose original contents are read from a file. The zeroed page option is referred to as demand zero.

Backing Storage for Global Sections Global section pages require backing storage on disk so that more frequently referenced code or data pages can occupy physical memory. The paging of least recently used pages is typical of a virtual memory system. The backing storage for a global section can be the system page files, a file opened by OpenVMS, or a file opened by the application. A global section backed by system page files is referred to as a page-file-backed global section. A global section backed by a specified file is referred to as a file-backed global section.

When a global section page is invalid in all process PTEs, the page is eligible to be written to an on-disk backing storage file. The physical page may remain in memory on a list of modified or free pages. OpenVMS algorithms and system dynamics, however, determine which page is written to disk.

Process Working Sets

On OpenVMS, a process' valid memory is tracked within its working set lists. The working set of a process reflects the amount of physical memory a process is consuming at one particular point in time. Each valid working set list entry represents one page of virtual memory whose corresponding process PTE is valid. A process' working set list includes global section pages, process private section pages, process private code pages, stack pages, and page table pages.

A process' working set quota is limited to 512 MB and sets the upper limit on the number of pages that can be swapped to disk. The limit on working set quota matches the size of a swap I/O request.[2] The effects on swapping would have to be examined to increase working set quotas above 512 MB.

Process working set lists are kept in 32-bit system address space. When free memory is plentiful in the system, process working set lists can increase to an extended quota specified in the system's account file for the user. The system parameter, WSMAX, specifies the maximum size to which a process working set can be extended. OpenVMS specifies an absolute maximum value of 4 GB for the WSMAX system parameter. An inverse relationship exists between the size specified for WSMAX and the number of resident processes OpenVMS can support, since both are maintained in the 32-bit addressable portion of system space. For example, specifying the maximum value for WSMAX sharply decreases the number of resident processes that can be specified.

Should OpenVMS be required to support larger working sets in the future, the working set lists would have to be moved out of 32-bit system space.

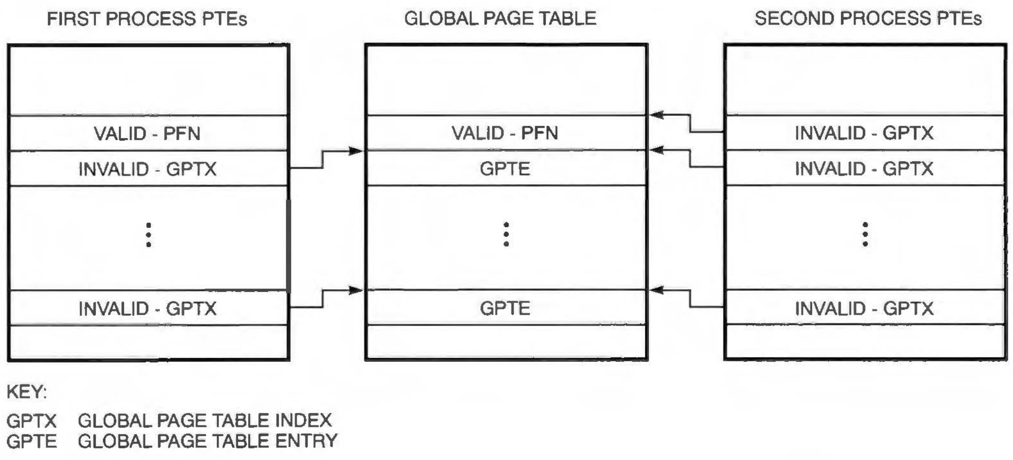


Figure 2
Process and Global PTEs

Page Fault Handling for Global Section Pages

The data within a global section may be heavily accessed by the many processes that are sharing the data. Therefore, the access time to the global section pages may influence the overall performance of the application.

Many hardware and software factors can influence the speed at which a page within a global section is accessed by a process. The factors relevant to this discussion are the following:

- Is the process PTE valid or invalid?
- If the process PTE is invalid, is the global section page valid or invalid?
- If the global section page is invalid, is the page on the modified list, free page list, or on disk within the backing storage file?

If the process PTE is invalid at the time the page is accessed, a translation invalid fault, or page fault, is generated by the hardware. The OpenVMS page fault handler determines the steps necessary to make the process PTE valid.

If the global section page is valid, the PFN of the data is read from the global section data structures. This is called a global valid fault. This type of fault is corrected quickly because the data that handles this fault is readily accessible from the data structures in memory.

If the global section page is invalid, the data may still be within a physical page on the modified or free page list maintained by OpenVMS. To correct this type of fault, the PFN that holds the data must be removed from the modified or free page list, and the global section page must be made valid. Then the fault can be handled as if it were a global valid fault.

If the page is on disk within the backing storage file, an I/O operation must be performed to read the data from the disk into memory before the global section page and process PTE can be made valid. This is the slowest type of global page fault, because performing a read I/O operation is much slower than manipulating data structures in memory.

For an application to experience the most efficient access to its shared pages, its process PTEs should be kept valid. An application may use system services to lock pages in the working set or in memory, but typically the approach taken by applications to reduce page fault overhead is to increase the user account's working set quota. This approach does not work when the size of the global section data exceeds the size of the working set quota limit of 512 MB.

Database Caches as File-backed Global Sections

Quick access to a database application's shared memory is critical for an application to handle transactions quickly.

Global sections implement shared memory on OpenVMS, so that many database processes can share the cached database records. Since global sections must have backing storage on disk, database caches are either backed by the system's page files or by a file created by the database application.

For best performance, the database application should keep all its global section pages valid in the process PTEs to avoid page fault and I/O overhead. Database processes write modified buffers from the cache to the database files on an as-needed basis. Therefore, the backing storage file required by OpenVMS is redundant storage.

Very Large Global Sections

The OpenVMS VLM project focused on VLM database cache design. An additional goal was to design the VLM features so that other types of VLM applications could benefit as well.

Consider a database cache that is 6 GB in size. Global sections of this magnitude are supported on OpenVMS Alpha with 64-bit addressing support. If the system page files are not used, the application must create and open a 6-GB file to be used as backing storage for the global section.

With the maximum quota of 512 MB for a process working set and with the maximum of a 4-GB working set size, no process could keep the entire 6-GB database cache valid in its working set at once. When an OpenVMS global section is used to implement the database cache, page faults are inevitable. Page fault activity severely impacts the performance of the VLM database cache by causing unnecessary I/O to and from the disk while managing these pages.

Since all global sections are pageable, a 6-GB file needs to be created for backing storage purposes. In the ideal case, the backing storage file is never used. The backing storage file is actually redundant with the database files themselves.

VLM Design Areas

The VLM design team targeted very large global sections (4 GB or larger) to share data among many processes. Furthermore, we assumed that the global section's contents would consist of zeroed memory instead of originating from a file. The team explored whether this focus was too narrow. We were concerned that implementing just one type of VLM global section would preclude support for certain types of VLM applications.

We considered that VLM applications might use very large amounts of memory whose contents originate from a data file. One type of read-only data from a file contains program instructions (or code). Code sections are currently not pushing the limits of 32-bit address space. Another type of read-only data from a file contains scientific data to be analyzed by the VLM application. To accommodate very large read-only data of this type, a large zeroed global section can be created, the data from the file can be read into memory, and then the data can be processed in memory.

If writable pages are initially read from a file instead of zeroed, the data will most likely need to be written back to the original file. In this case, the file can be used as the backing storage for the data. This type of VLM global section is supported on OpenVMS Alpha as a file-backed global section. The operating system's algorithm for working set page replacement keeps the most recently accessed pages in memory. Working set quotas greater than 512 MB and working set sizes greater than 4 GB help this type of VLM application scale to higher memory sizes.

We also considered very large demand-zero private pages, "malloc" or "heap" memory. The system page files are the backing storage for demand-zero private pages. Currently, processes can have a page file quota as large as 32 GB. A VLM application, however, may not want these private data pages to be written to a page file since the pages are used in a similar fashion as in-memory caches. Larger working set quotas also help this type of VLM application accommodate ever-increasing memory sizes.

Backing Storage Issues

For many years, database cache designers and database performance experts had requested that the OpenVMS operating system support memory with no backing storage files. The backing storage was not only redundant but also wasteful of disk space. The waste issue is made worse as the sizes of the database caches approach the 4-GB range. As a result, the OpenVMS Alpha VLM design had to allow for non-file-backed global sections.

The support of 64-bit addressing and VLM has always been viewed as a two-phased approach, so that functionality could be delivered in a timely fashion.[3] OpenVMS Alpha version 7.0 provided the essentials of 64-bit addressing support. The VLM support was viewed as an extension to the memory management model and was deferred to OpenVMS Alpha version 7.1.

Working Set List Issues. Entries in the process working set list are not required for pages that can never be written to a backing storage file. The fundamental concept of the OpenVMS working set algorithms is to support the paging of data from memory to disk and back into memory when it is needed again. Since the focus of the VLM design was on memory that would not be backed by disk storage, the VLM design team realized that these pages, although valid in the process PTEs, did not need to be in the process' working set list.

VLM Programming Interface

The OpenVMS Alpha VLM design provides a new programming interface for VLM applications to create, map to, and delete demand-zero, memory-resident global sections. The existing programming interfaces did not easily accommodate the new VLM features.

To justify a new programming interface, we looked at the applications that would be calling the new system service routines. To address more than 4 GB of memory in the flat OpenVMS 64-bit address space, a 32-bit application must be recompiled to use 64-bit pointers and often requires source code changes as well. Database applications were already modifying their source code to use 64-bit pointers and to scale their algorithms to handle VLM systems.[1] Therefore, calling a new set of system service routines was considered acceptable to the programmers of VLM applications.

Options for Memory-resident Global Sections

To initialize a very large memory-resident global section mapped by several processes, the overhead of hardware faults, allocating zeroed pages, setting process PTEs valid, and setting global section pages valid is eliminated by preallocating the physical pages for the memory-resident global section. Preallocation is performed by the reserved memory registry, and is discussed later in this paper. Here we talk about options for how the reserved memory is used.

Two options, ALLOC and FLUID, are available for creating a demand-zero, memory-resident global section.

ALLOC Option The ALLOC option uses preallocated, zeroed pages of memory for the global section. When the ALLOC option is used, pages are set aside during system start-up specifically for the memory-resident global section. Preallocation of contiguous groups of pages is discussed in the section Reserving Memory during System Start-up. Preallocated memory-resident global sections are faster to initialize than memory-resident global sections that use the FLUID option.

Run-time performance is improved by using the Alpha Architecture's granularity hint, a mechanism we discuss later in this paper. To use the ALLOC option, the system must be rebooted for large ranges of physically contiguous memory to be allocated.

FLUID Option The FLUID option allows pages not yet accessed within the global section to remain fluid within the system. This is also referred to as the fault option because the page fault algorithm is used to allocate the pages. When the FLUID (or fault) option is used, processes or the system can use the physical pages until they are accessed within the memory-resident global section. The pages remain within the system's fluid memory until they are needed. This type of memory-resident global section is more flexible than one that uses the ALLOC option. If an application that uses a memory-resident global section is run on a system that cannot be rebooted due to system availability concerns, it can still use the FLUID option. The system will not allow this application to run unless enough pages of memory are available in the system for the memory-resident global section.

The system service internals code checks the reserved memory registry to determine the range of pages preallocated for the memory-resident global section or to determine if the FLUID option will be used. Therefore the decision to use the ALLOC or the FLUID option is not made within the system services routine interface. The system manager can determine which option is used by specifying preferences in the reserved memory registry. An application can be switched from using the ALLOC option to using the FLUID option without requiring a system reboot.

Design Internals

The internals of the design choices underscore the modularity of the shared memory model using global sections. A new global section type was easily added to the OpenVMS system. Those aspects of memory-resident global sections that are identical to pageable global sections required no code modifications to support.

To support memory-resident global sections, the MRES and ALLOC flags were added to the existing global section data structures. The MRES flag indicates that the global section is memory resident, and the ALLOC flag indicates that contiguous pages were preallocated for the global section.

The file-backing storage information within global section data structures is set to zero for memory-resident global sections to indicate that no backing storage file is used. Other than the new flags and the lack of backing storage file information, a demand-zero, memory-resident global section looks to OpenVMS Alpha memory management like a demand-zero, file-backed global section. Figure 3 shows the updates to the global section data structures.

One important difference with memory-resident global sections is that once a global section page becomes valid, it remains valid for the life of the global section. Global section pages by definition can never become invalid for a memory-resident global section.

When a process maps to a memory-resident global section, the process PTE can be either valid for the ALLOC option or invalid for the FLUID option. When the ALLOC option is used, no page faulting occurs for the global section pages.

When a process first accesses an invalid memory-resident global section page, a page fault occurs just as with traditional file-backed global sections. Because the same data structures are present, the page fault code initially executes the code for a demand-zero, file-backed global section page. A zeroed page is allocated and placed in the global section data structures, and the process PTE is set valid.

The working set list manipulation steps are skipped when the MRES flag is encountered in the global section data structures. Because these global section pages are not placed in the process working set list, they are not considered in its page-replacement algorithm. As such, the OpenVMS Alpha working set manipulation code paths remained unchanged.

System Management and Memory-resident Global Sections

When a memory-resident global section is used instead of a traditional, pageable global section for a database cache, there is no longer any wasted page file storage required by OpenVMS to back up the global section.

The other system management issue alleviated by the implementation of memory-resident global sections concerns working set sizes and quotas. When a file-backed global section is used for the database cache, the database processes require elevated working

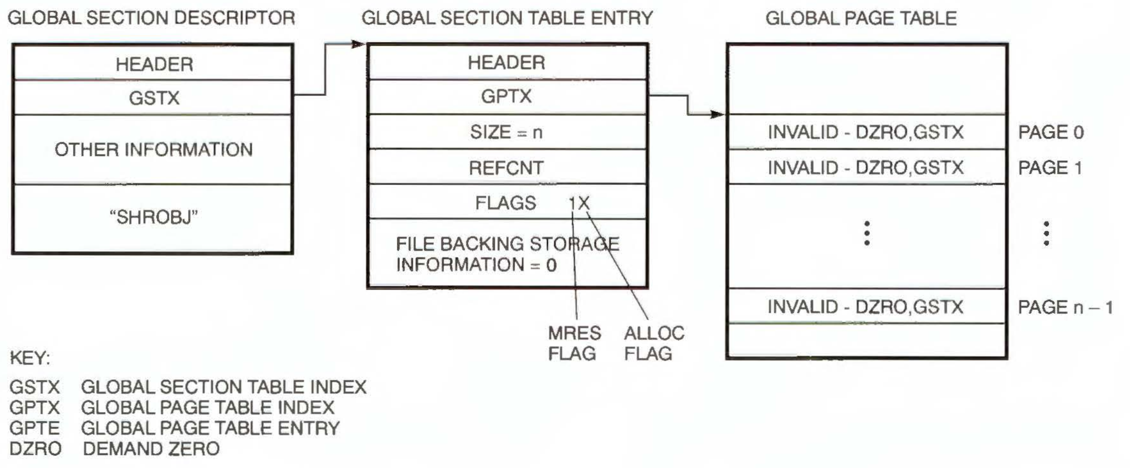


Figure 3
Memory-resident Global Section Data Structures

set quotas to accommodate the size of the database cache. This is no longer a concern because memory-resident global section pages are not placed into the process working set list.

With the use of memory-resident global sections, system managers may reduce the value for the WSMAX system parameter such that more processes can remain resident within the system. Recall that a process working set list is in 32-bit system address space, which is limited to 2 GB.

Shared Page Tables

VLM applications typically consume large amounts of physical memory in an attempt to minimize disk I/O and enhance overall application performance. As the physical memory requirements of VLM applications increase, the following second-order effects are observed due to the overhead of mapping to very large global sections:

- Noticeably long application start-up and shut-down times
- Additional need for physical memory as the number of concurrent sharers of a large global section increases
- Unanticipated exhaustion of the working set quota and page file quota
- A reduction in the number of processes resident in memory, resulting in increased process swapping

The first two effects are related to page table mapping overhead and size. The second two effects, as they relate to page table quota accounting, were also resolved by a shared page tables implementation. The following sections address the first two issues since they uniquely pertain to the page table overhead.

Application Start-up and Shut-down Times

Users of VLM applications can observe long application start-up and shut-down times as a result of creating and deleting very large amounts of virtual memory. A single process mapping to a very large virtual memory object does not impact overall system performance. However, a great number of processes that simultaneously map to a very large virtual memory object have a noticeable impact on the system's responsiveness. The primary cause of the performance degradation is the accelerated contention for internal operating system locks. This observation has been witnessed on OpenVMS systems and on DIGITAL UNIX systems (prior to the addition of VLM support.)

On OpenVMS, the memory management spinlock (a synchronization mechanism) serializes access to privileged, memory-management data structures. We have observed increased spinlock contention as the result of hundreds of processes simultaneously mapping to large global sections. Similar lock contention and system unresponsiveness occur when multiple processes attempt to delete their address space simultaneously.

Additional Need for Physical Memory

For pages of virtual memory to be valid and resident, the page table pages that map the data pages must also be valid and resident. If the page table pages are not in memory, successful address translation cannot occur.

Consider an 8-GB, memory-resident global section on an OpenVMS Alpha system (with an 8-kilobyte page size and 8-byte PTE size). Each process that maps the entire 8-GB, memory-resident global section requires 8 MB for the associated page table structures. If 100 processes are mapping the memory-resident global section, an additional 800 MB of physical memory must be available to accommodate all processes' page table structures. This further requires that working set list sizes, process page file quotas, and system page files be large enough to accommodate the page tables.

When 100 processes are mapping to the same memory-resident global section, the same PTE data is replicated into the page tables of the 100 processes. If each process could share the page table data, only 8 MB of physical memory would be required to map an 8-GB, memory-resident global section; 792 MB of physical memory would be available for other system purposes.

Figure 4 shows the amount of memory used for process page tables mapping global sections ranging in size from 2 to 8 GB. Note that as the number of processes that map an 8-GB global section exceeds

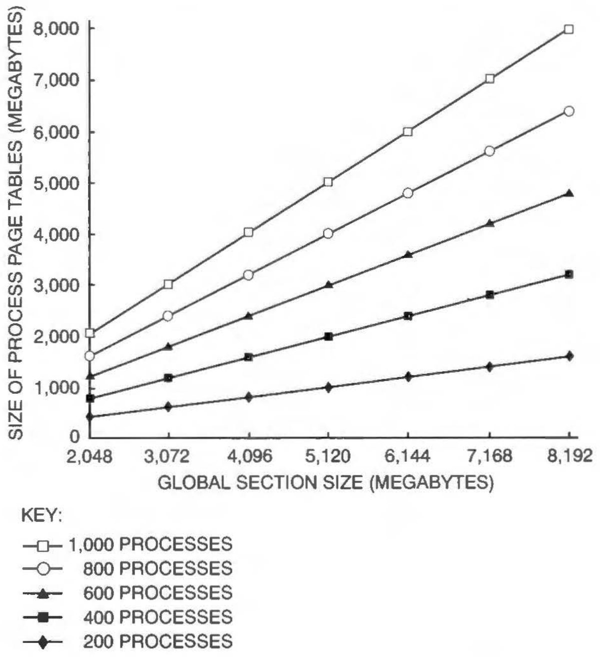


Figure 4
Process Page Table Sizes

1,000, the amount of memory used by process page tables is larger than the global section itself.

Shared Memory Models

We sought a solution to sharing process page tables that would alleviate the performance problems and memory utilization overhead yet stay within the shared memory framework provided by the operating system and the architecture. Two shared memory models are implemented on OpenVMS, shared system address space and global sections.

The OpenVMS operating system supports an address space layout that includes a shared system address space, page table space, and private process address space. Shared system address space is created by placing the physical address of the shared system space page tables into every process' top-level page table. Thus, every process has the same lower-level page tables in its virtual-to-physical address translation path. In turn, the same operating system code and data are found in all processes' address spaces at the same virtual address ranges. A similar means could be used to create a shared page table space that is used to map one or more memory-resident global sections.

An alternative for sharing the page tables is to create a global section that describes the page table structure. The operating system could maintain the association between the memory-resident global section and the global section for its shared page table pages. The shared page table global section could be mapped at the upper levels of the table structure such that each process that maps to it has the same lower-level page tables in its virtual-to-physical address translation path. This in turn would cause the data to be mapped by all the processes.

Figure 5 provides a conceptual representation of the shared memory model. Figure 6 extends the shared memory model by demonstrating that the page tables become a part of the shared memory object.

The benefits and drawbacks of both sharing models are highlighted in Table 1 and Table 2.

Model Chosen for Sharing Page Tables

After examining the existing memory-sharing models on OpenVMS and taking careful note of the composition and characteristics of shared page tables, the design team chose to implement shared page tables as a global section. In addition to the benefits listed in Table 2, the

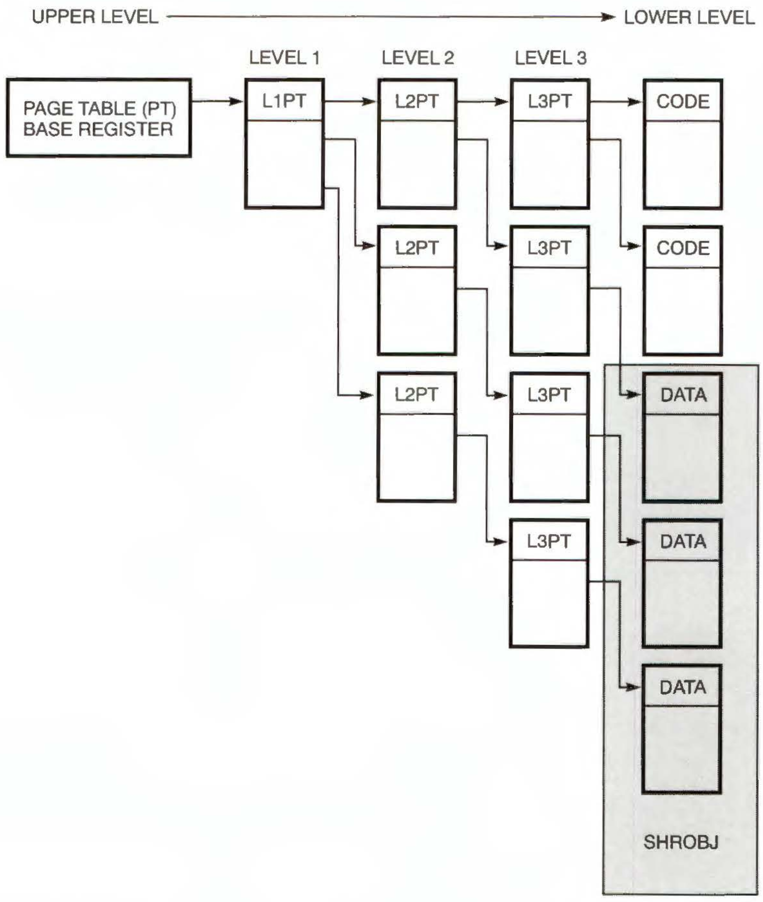


Figure 5
Shared Memory Object

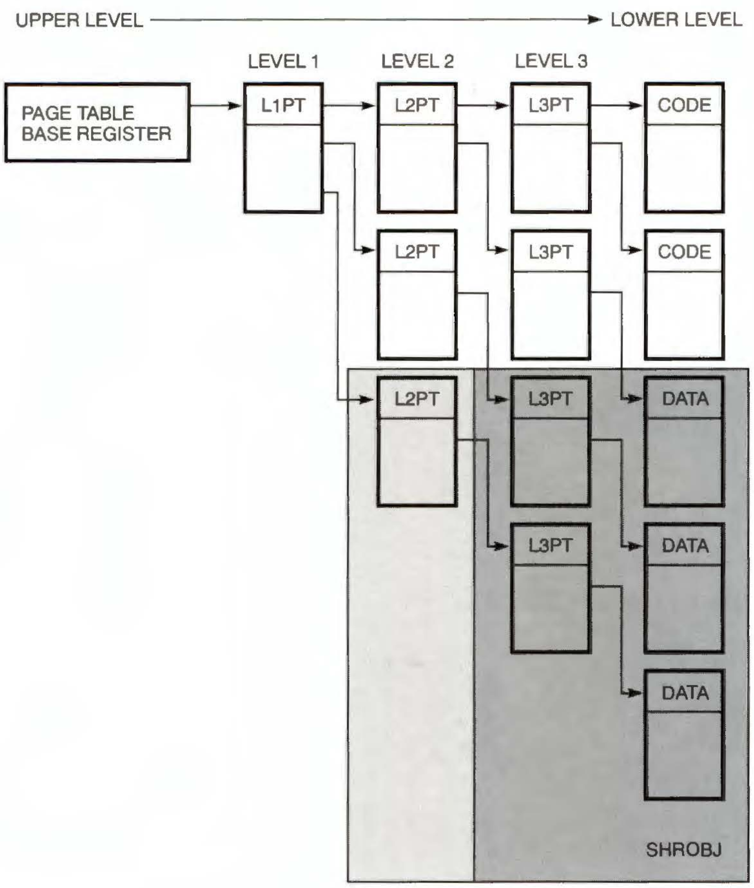


Figure 6
Shared Memory Objects Using Shared Page Tables

Table 1
Shared Page Table Space—Benefits and Drawbacks

Benefits	Drawbacks
Shared page table space begins at the same virtual address for all processes.	The virtual address space is reserved for every process. Processes not using shared page tables are penalized by a loss in available address space. Shared page table space is at least 8 GB in size, regardless of whether the entire space is used. A significant amount of new code would need to be added to the kernel since shared system space is managed separately from process address space.

Table 2
Global Sections for Page Tables—Benefits and Drawbacks

Benefits	Drawbacks
The same virtual addresses can be used by all processes, but this is not required. The amount of virtual address space mapped by shared page tables is determined by application need. Shared page tables are available only to those processes that need them. Shared page tables allow for significant reuse of existing global section data structures and process address space management code.	Shared page tables are mapped at different virtual addresses per process unless additional steps are taken.

design team noticed that shared page table pages bear great resemblance to the memory-resident pages they map. Specifically, for a data or code page to be valid and resident, its page table page must also be valid and resident. The ability to reuse a significant amount of the global section management code reduced the debugging and testing phases of the project.

In the initial implementation, shared page table global sections map to memory-resident global sections only. This decision was made because the design focused on the demands of VLM applications that use memory-resident global sections. Should significant demand exist, the implementation can be expanded to allow the mapping of pageable global sections.

Shared page tables can never map process private data. The design team had to ensure that the shared page table implementation kept process private data from entering a virtual address range mapped by a shared page table page. If this were to happen, it would compromise the security of data access between processes.

Shared Page Tables Design

The goals for the design of shared page tables included the following:

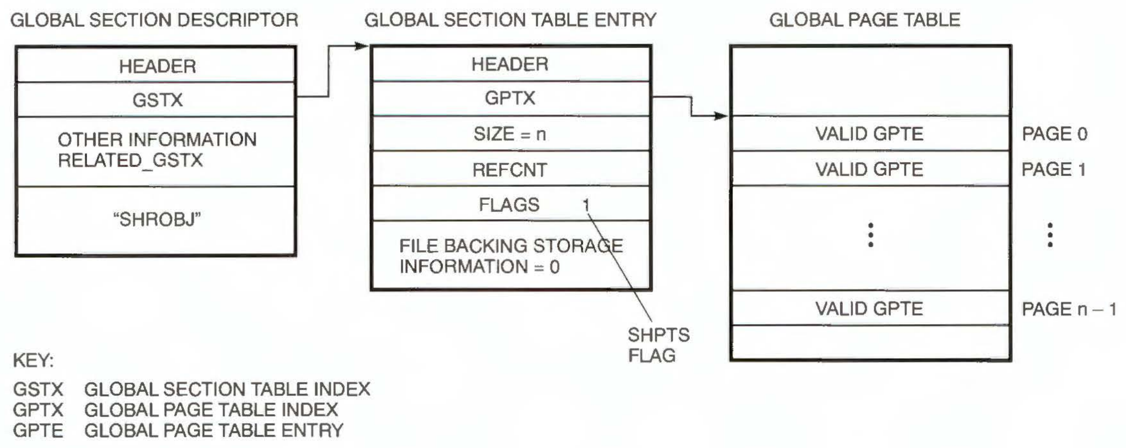
- Reduce the time required for multiple users to map the same memory-resident global section
- Reduce the physical memory cost of maintaining private page tables for multiple mappers of the same memory-resident global section
- Do not require the use of a backing storage file for shared page table pages
- Eliminate the working set list accounting for these page table pages
- Implement a design that allows upper levels of the page table hierarchy to be shared at a later time

Figure 6 demonstrates the shared page table global section model. The dark gray portion of the figure highlights the level of sharing supplied in OpenVMS Alpha version 7.1. The light gray portion highlights possible levels of sharing allowed by creating a shared page table global section consisting of upper-level page table pages.

Modifications to Global Section Data Structure Table 2 noted as a benefit the ability to reuse existing data structures and code. Minor modifications were exacted to the global section data structures so that they could be used to represent a shared page table global section. A new flag, SHARED_PTS, was added to the global section data structures. Coupled with this change was the requirement that a memory-resident global section and its shared page table global section be uniquely linked together. The correspondence between the two sets of global sections is managed by the operating system and is used to locate the data structures for one global section when the structures for the other global section are in hand. Figure 7 highlights the changes made to the data structures.

Creating Shared Page Tables To create a memory-resident global section, an application calls a system service routine. No flags or extra arguments are required to enable the creation of an associated shared page table global section.

The design team also provided a means to disable the creation of the shared page tables in the event that a user might find shared page tables to be undesirable. To disable the creation of shared page tables, the reserved memory registry entry associated with the memory-resident global section can specify that page tables are not to be used. Within the system service routine that creates a memory-resident global section,



GLOBAL SECTION DESCRIPTOR
HEADER
GSTX
OTHER INFORMATION RELATED_GSTX
"SHROBJ"
GLOBAL SECTION TABLE ENTRY
HEADER
GPTX
SIZE = n
REFCNT
FLAGS 1
FILE BACKING STORAGE INFORMATION = 0
SHPTS FLAG
GLOBAL PAGE TABLE
VALID GPTE
VALID GPTE
VALID GPTE
PAGE 0
PAGE 1
PAGE n – 1
KEY:
GSTX GLOBAL SECTION TABLE INDEX
GPTX GLOBAL PAGE TABLE INDEX
GPTE GLOBAL PAGE TABLE ENTRY

Figure 7
Data Structure Modifications

the reserved memory registry is examined for an entry associated with the named global section. If an entry exists and it specifies shared page tables, shared page tables are created. If the entry does not specify shared page tables, shared page tables are not created.

If no entry exists for the global section at all, shared page tables are created. Thus, shared page tables are created by default if no action is taken to disable their creation. We believed that most applications would benefit from shared page tables and thus should be created transparently by default.

Once the decision is made to create shared page tables for the global section, the system service routine allocates a set of global section data structures for the shared page table global section. These structures are initialized in the same manner as their memory-resident counterparts, and in many cases the fields in both sets of structures contain identical data.

Note that on current Alpha platforms, there is one shared page table page for every 1,024 global section pages or 8 MB. (The number of shared page table pages is rounded up to accommodate global sections that are not even multiples of 8 MB in size.)

Shared PTEs represent the data within a shared page table global section and are initialized by the operating system. Since page table pages are not accessible through page table space[4] until the process maps to the data, the initialization of the shared page table pages presented some design issues. To initialize the shared page table pages, they must be mapped, yet they are not mapped at the time that the global section is created.

A simple solution to the problem was chosen. Each shared page table page is temporarily mapped to a system space virtual page solely for the purposes of initializing the shared PTEs. Temporarily mapping each page allows the shared page table global section to be fully initialized at the time it is created.

An interesting alternative for initializing the pages would have been to set the upper-level PTEs invalid, referencing the shared page table global section. The page fault handler could initialize a shared page table page when a process accesses a global section page, thus referencing an invalid page table page. The shared page table page could then be initialized through its mapping in page table space. Once the page is initialized and made valid, other processes referencing the same data would incur a global valid fault for the shared page table page. This design was rejected due to the additional overhead of faulting during execution of the application, especially when the ALLOC option is used for the memory-resident global section.

Mapping to a Shared Page Table Global Section Mapping to a memory-resident global section that has shared page tables presented new challenges and constraints on the mapping criteria normally imposed by the virtual address space creation routines. The mapping service routines require more stringent mapping criteria when mapping to a memory-resident global section that has shared page tables. These requirements serve two purposes:

1. Prevent process private data from being mapped onto shared page tables. If part of a shared page table page is unused because the memory-resident global section is not an even multiple of 8 MB, the process would normally be allowed to create private data there.
2. Accommodate the virtual addressing alignments required when mapping page tables into a process' address space.

For applications that cannot be changed to conform to these mapping restrictions, a memory-resident global section with shared page tables can be mapped using the process' private page tables. This capability is also useful when the memory-resident global section is mapped read-only. This mapping cannot share page tables with a writable mapping because the access protection is stored within the shared PTEs.

Shared Page Table Virtual Regions The virtual region support added in OpenVMS Alpha version 7.0 was extended to aid in prohibiting process private pages from being mapped by PTEs within shared page tables. Virtual regions are lightweight objects a process can use to reserve portions of its process virtual address space. Reserving address space prevents other threads in the process from creating address space in the reserved area, unless they specify the handle of that reserved area to the address space creation routines.

To control which portion of the address space is mapped with shared page tables, the shared page table attribute was added to virtual regions. To map a memory-resident global section with shared page tables, the user must supply the mapping routine with the name of the appropriate global section and the region handle of a shared page table virtual region.

There are two constraints on the size and alignment of shared page table virtual regions.

1. The size of a shared page table virtual region must be an even multiple of bytes mapped by a page table page. For an 8-KB page system, the size of any shared page table virtual region is an even multiple of 8 MB.
2. The caller can specify a particular starting virtual address for a virtual region. For shared page table virtual regions, the starting virtual address must be aligned to an 8-MB boundary. If the operating system chooses the virtual address for the region, it ensures the virtual address is properly aligned.

If either the size or the alignment requirement for a shared page table virtual region is not met, the service fails to create the region.

The size and alignment constraints placed on shared page table virtual regions keep page table pages from spanning two different virtual regions. This allows the operating system to restrict process private mappings in shared page table regions and shared page table mappings in other regions by checking the shared page table's attribute of the region before starting the mapping operation.

Mapping within Shared Page Table Regions The address space mapped within a shared page table virtual region also must be page table page aligned. This ensures that mappings to multiple memory-resident global sections that have unique sets of shared page tables do not encroach upon each other.

The map length is the only argument to the mapping system service routines that need not be an even multiple of bytes mapped by a page table page. This is allowed because it is possible for the size of the memory-resident global section to not be an even multiple of bytes mapped by a page table page. A memory-resident global section that fits this length description will have a portion of its last shared page table page unused.

The Reserved Memory Registry

OpenVMS Alpha VLM support provides a physical memory reservation system that can be exploited by VLM applications. The main purpose of this system is to provide portions of the system's physical memory to multiple consumers. When necessary, a consumer can reserve a quantity of physical addresses in an attempt to make the most efficient use of system components, namely the translation buffer. More efficient use of the CPU and its peripheral components leads to increased application performance.

Alpha Granularity Hint Regions

A translation buffer (TB) is a CPU component that caches recent virtual-to-physical address translations of valid pages. The TB is a small amount of very fast memory and therefore is only capable of caching a limited number of translations. Each entry in the TB represents a single successful virtual-to-physical address translation. TB entries are purged either when a request is made by software or when the TB is full and a more recent translation needs to be cached.

The Alpha Architecture coupled with software can help make more effective use of the TB by allowing several contiguous pages (groups of 8, 64, or 512) to act as a single huge page. This single huge page is called a granularity hint region and is composed of contiguous virtual and physical pages whose respective first pages are exactly aligned according to the number of pages in the region. When the conditions for a granularity hint region prevail, the single huge page is allowed to consume a single TB entry instead of several. Minimizing the number of entries consumed for contiguous pages greatly reduces turnover within the TB, leading to higher chances of a TB hit. Increasing the likelihood of a TB hit in turn minimizes the number of virtual-to-physical translations performed by the CPU.[5]

Since memory-resident global sections are nonpageable, mappings to memory-resident global sections greatly benefit by exploiting granularity hint regions. Unfortunately, there is no guarantee that a contiguous set of physical pages (let alone pages that meet the alignment criteria) can be located once the system is initialized and ready for steady-state operations.

Limiting Physical Memory

One technique to locate a contiguous set of PFNs on OpenVMS (previously used on Alpha and VAX platforms) is to limit the actual number of physical pages used by the operating system. This is accomplished by setting the PHYSICAL_MEMORY system parameter to a value smaller than the actual amount of physical memory available in the system. The system is then rebooted, and the PFNs that represent higher physical addresses than that specified by the parameter are allocated by the application.

This technique works well for a single application that wishes to allocate or use a range of PFNs not used by the operating system. Unfortunately, it suffers from the following problems:

- It requires the application to determine the first page not used by the operating system.
- It requires a process running this application to be highly privileged since the operating system does not check which PFNs are being mapped.
- Since the operating system does not arbitrate access to the isolated physical addresses, only one application can safely use them.
- The Alpha Architecture allows for implementations to support discontiguous physical memory or physical memory holes. This means that there is no guarantee that the isolated physical addresses are successively adjacent.
- The PFNs above the limit set are not managed by the operating system (physical memory data structures do not describe these PFNs). Therefore, the pages above the limit cannot be reclaimed by the operating system once the application is finished using them unless the system is rebooted.

The Reserved Memory Solution

The OpenVMS reserved memory registry was created to provide contiguous physical memory for the purposes of further improving the performance of VLM applications. The reserved memory registry allows the system manager to specify multiple memory reservations based on the needs of various VLM applications.

The reserved memory registry has the ability to reserve a preallocated set of PFNs. This allows a set of contiguous pages to be preallocated with the appropriate alignment to allow an Alpha granularity hint region to be created with the pages. It can also reserve physical memory that is not preallocated. Effectively, the application creating such a reservation can allocate the pages as required. The reservation ensures that the system is tuned to exclude these pages.

The reserved memory registry can specify a reservation consisting of prezeroed PFNs. It can also specify that a reservation account for any associated page tables. The reservation system allows the system manager to free a reservation when the corresponding consumer no longer needs that physical memory.

The memory reserved by the reserved memory registry is communicated to OpenVMS system tuning facilities such that the deduction in fluid memory is noted when computing system parameters that rely on the amount of physical memory in the system.

SYSMAN User Interface The OpenVMS Alpha SYSMAN utility supports the RESERVED_MEMORY command for manipulating entries in the reserved memory registry. A unique character string is specified as the entry's handle when the entry is added, modified, or removed. A size in megabytes is specified for each entry added.

Each reserved memory registry entry can have the following options: preallocated PFNs (ALLOC), zeroed PFNs, and an allotment for page tables. VLM applications enter their unique requirements for reserved memory. For memory-resident global sections, zeroed PFNs and page tables are usually specified.

Reserving Memory during System Start-up To ensure that the contiguous pages can be allocated and that run-time physical memory allocation routines can be used, reserved memory allocations occur soon after the operating system's physical memory data structures have been initialized.

The reserved memory registry data file is read to begin the reservation process. Information about each entry is stored in a data structure. Multiple entries result in multiple structures being linked together in a descending-order linked list. The list is intentionally ordered in this manner, so that the largest reservations are honored first and contiguous memory is not fragmented with smaller requests.

For entries with the ALLOC characteristic, an attempt is made to locate pages that will satisfy the largest granularity hint region that fits within the request. For example, reservation requests that are larger than 4 MB result in the first page allocated to be aligned to meet the requirements of a 512-page granularity hint region.

The system's fluid page counter is reduced to account for the amount of reserved memory specified in each entry. This counter tracks the number of physical pages that can be reclaimed from processes or the system through paging and swapping. Another system-defined value, minimum fluid page count, is calculated during system initialization and represents the absolute minimum number of fluid pages the system needs to function. Deductions from the fluid page count are always checked against the minimum fluid page count to prevent the system from becoming starved for pages.

Running AUTOGEN, the OpenVMS system tuning utility, after modifying the reserved memory registry allows for proper initialization of the fluid page counter, the minimum fluid page count, and other system parameters, thereby accommodating the change in reserved memory. AUTOGEN considers entries in the reserved memory registry before selecting values for system parameters that are based on the system's memory size. Failing to retune the system can lead to unbootable system configurations as well as poorly tuned systems.

Page Tables Characteristic The page table reserved memory registry characteristic specifies that the reserved memory allotment for a particular entry should include enough pages for its page table requirements. The reserved memory registry reserves enough memory to account for lower-level page table pages, although the overall design can accommodate allotments for page tables at any level.

The page table characteristic can be omitted if shared page tables are not desired for a particular memory-resident global section or if the reserved memory will be used for another purpose. For example, a privileged application such as a driver could call the kernel-mode reserved memory registry routines directly to use its reservation from the registry. In this case, page tables are already provided by the operating system since the reserved pages will be mapped in shared system address space.

Using Reserved Memory Entries are used and returned to the reserved memory registry using a set of kernel-mode routines. These routines can be called by applications running in kernel mode such as the system service routines that create memory-resident global sections. For an application to create a memory-resident global section and use reserved memory, the global section name must exactly match the name of the reserved memory registry entry.

After the system service routine has obtained the reserved memory for the memory-resident global section, it calls a reserved memory registry routine again for the associated shared page table global section. If page tables were not specified for the entry, the system service routine does not create a shared page table global section.

A side benefit of using the ALLOC option for the memory-resident global section is that the shared page tables can be mapped into page table space using granularity hint regions as well.

Returning Reserved Memory The memory used by a memory-resident global section and its associated shared page table global section is returned to the reserved memory registry (by calling a kernel-mode routine) when the global section is deleted. Reserved memory is only returned when a memory-resident global section has no more outstanding references. Preallocated pages are not returned to the system's free page list.

Freeing Reserved Memory Preallocated reserved memory that is unused or partially used can be freed to the system's free page list and added to the system's fluid page count. Reserved fluid memory is returned to the system's fluid page count only.

Once an entry's reserved memory has been freed, subsequent attempts to use reserved memory with the same name may be able to use only the FLUID option, because a preallocated set of pages is no longer set aside for the memory-resident global section. (If the system's fluid page count is large enough to accommodate the request, it will be honored.)

The ability to free unused or partially used reserved memory registry entries adds flexibility to the management of the system. If applications need more memory, the registry can still be run with the FLUID option until the system can be rebooted with a larger amount of reserved memory. A pool of reserved memory can be freed at system start-up so that multiple applications can use memory-resident global sections to a limit specified by the system manager in the reserved memory registry.

Reserved Memory Registry and Other Applications Other OpenVMS system components and applications may also be able to take advantage of the reserved memory registry.

Applications that relied upon modifications to the PHYSICAL_MEMORY system parameter as a means of gaining exclusive access to physical memory can enter kernel mode and call the reserved memory registry kernel-mode routines directly as an alternative. Once a contiguous range of PFNs is obtained, the application can map the pages as before.

Using and returning reserved memory registry entries requires kernel-mode access. This is not viewed as a problem because applications using the former method (of modifying the PHYSICAL_MEMORY system parameter) were already privileged. Using the reserved memory registry solves the problems associated with the previous approach and requires few code changes.

Performance Results

In a paper describing the 64-bit option for the Oracle7 Relational Database System,[1] the author underscores the benefits realized on a VLM system running the DIGITAL UNIX operating system. The test results described in that paper highlight the benefits of being able to cache large amounts of data instead of resorting to disk I/O. Although the OpenVMS design team was not able to execute similar kinds of product tests, we expected to realize similar performance improvements for the following reasons:

- More of a VLM application's hot data is kept resident instead of paging between memory and secondary storage.
- Application start-up and shut-down times are significantly reduced since the page table structures for the large shared memory object are also shared. The result is that many fewer page tables need to be managed and manipulated per process.
- Reducing the amount of PTE manipulations results in reduced lock contention when hundreds of processes map the large shared memory object.

As an alternative to product testing, the design team devised experiments that simulate the simultaneous start-up of many database server processes. The experiments were specifically designed to measure the scaling effects of a VLM system during application start-up, not during steady-state operation.

We performed two basic tests. In the first, we used a 7.5-GB, memory-resident global section to measure the time required for an increasing number of server processes to start up. All server processes mapped to the same memory-resident global section using shared page tables. The results shown in Figure 8 indicate that the system easily accommodated 300 processes. Higher numbers of processes run simultaneously caused increasingly large amounts of system stress due to the paging of other process data.

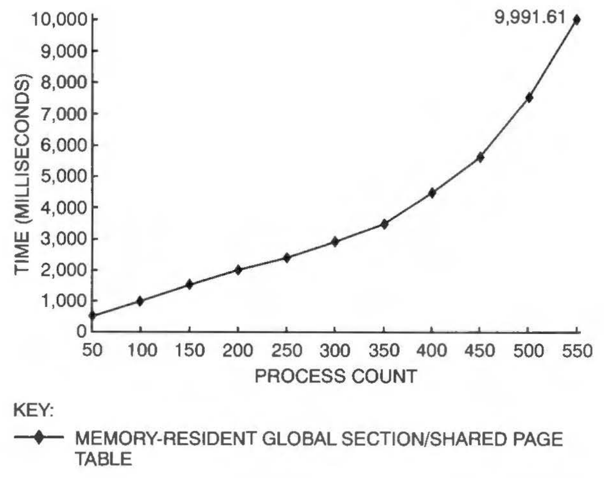


Figure 8
Server Start-up Time versus Process Count

In another test, we used 300 processes to measure the time required to map a memory-resident global section with and without shared page tables. In this test, the size of global section was varied. Note that the average time required to start up the server processes rises at nearly a constant rate when not using shared page tables. When the global section sizes were 5 GB and greater, the side effect of paging activity caused the start-up times to rise more sharply as shown in Figure 9.

The same was not true when using shared page tables. The time required to map the increasing section sizes remained constant at just under three seconds. The same experiment on an AlphaServer 8400 system with 28 GB of memory showed identical constant start-up times as the size of the memory-resident global section was increased to 27 GB.

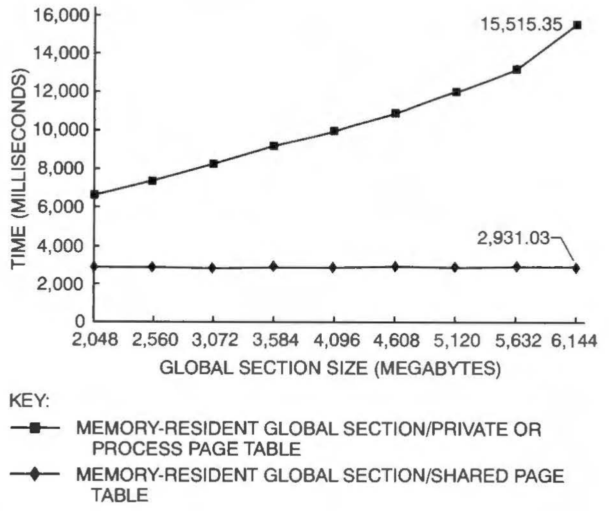


Figure 9
Server Start-up Time on an 8-GB System

Conclusion

The OpenVMS Alpha VLM support available in version 7.1 is a natural extension to the 64-bit virtual addressing support included in version 7.0. The 64-bit virtual addressing support removed the 4-GB virtual address space limit and allowed applications to make the most of the address space provided by Alpha systems. The VLM support enables database products or other applications that make significant demands on physical memory to make the most of large memory systems by allowing large caches to remain memory resident. The programming support provided as part of the VLM enhancements enables applications to take advantage of both 64-bit virtual addressing and very large memories in a modular fashion consistent with the OpenVMS shared memory model. This combination enables applications to realize the full power of Alpha VLM systems.

The Oracle7 Relational Database Management System for OpenVMS Alpha was modified by Oracle Corporation to exploit the VLM support described in this paper. The combination of memory-resident global sections, shared page tables, and the reserved memory registry has not only improved application start-up and run-time performance, but it has also simplified the management of OpenVMS Alpha VLM systems.

Acknowledgments

The authors wish to thank other members of the OpenVMS Alpha Kernel Group who contributed to the design and implementation of VLM support on OpenVMS Alpha: Tom Benson, Richard Bishop, Clair Grant, Andy Kuehnel, and Dave Wall. Jef Kennedy and Tony Lekas of Oracle Corporation provided great assistance as well. We would also like to thank Ron Higgins for his management support and Mike Harvey for his long-standing mentorship. Margie Sherlock and Tom Benson were especially helpful in assisting us with the writing aspects of this paper. We are also thankful to Gaitan D'Antoni who found the time in current project schedules to accommodate the writing of this paper.

References

1. V. Gokhale, "Design of the 64-bit Option for the Oracle7 Relational Database Management System," *Digital Technical Journal,* vol. 8, no. 4 (1996): 76–82.

2. R. Goldenberg and S. Saravanan, *OpenVMS AXP Internals and Data Structures, Version 1.5* (Newton, Mass.: Digital Press, 1994).

3. T. Benson, K. Noel, and R. Peterson, "The OpenVMS Mixed Pointer Sized Environment," *Digital Technical Journal,* vol. 8, no. 2 (1996): 72–82.

4. M. Harvey and L. Szubowicz, "Extending OpenVMS for 64-bit Addressable Virtual Memory," *Digital Technical Journal,* vol. 8, no. 2 (1996): 57–71.

5. R. Sites and R. Witek, *Alpha AXP Architecture Reference Manual,* 2d ed. (Newton, Mass.: Digital Press, 1995).

General References

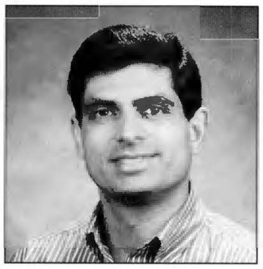
OpenVMS Alpha Guide to 64-bit Addressing and VLM Features (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBCB-TE).

OpenVMS System Services Reference Manual: A-GETMSG (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBMG-TE) and *OpenVMS System Services Reference Manual: GETQUI-Z* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBNB-TE).

For a more complete description of shared memory creation on DIGITAL UNIX, see the *DIGITAL UNIX Programmer's Guide.*

Biographies

Karen L. Noel
A consulting engineer in the OpenVMS Engineering Group, Karen Noel was the technical leader for the VLM project. Currently, as a member of the OpenVMS Galaxy design team, Karen is contributing to the overall Galaxy architecture and focusing on the design of memory partitioning and shared memory support. After receiving a B.S. in computer science from Cornell University in 1985, Karen joined DIGITAL's RSX Development Group. In 1990, she joined the VMS Group and ported several parts of the VMS kernel from the VAX platform to the Alpha platform. As one of the principal designers of OpenVMS Alpha 64-bit addressing and VLM support, she has applied for nine software patents.



Nitin Y. Karkhanis
Nitin Karkhanis joined DIGITAL in 1987. As a member of the OpenVMS Alpha Executive Group, he was one of the developers of OpenVMS Alpha 64-bit addressing support. He was also the primary developer for the physical memory hole (discontiguous physical memory) support for OpenVMS Alpha. He is a coapplicant for two patents on OpenVMS VLM. Currently, Nitin is a principal software engineer in the OpenVMS NT Infrastructure Engineering Group, where he is a member of the OpenVMS DCOM porting team. Nitin received a B.S. in computer science from the University of Vermont in 1987.

PowerStorm 4DT: A High-performance Graphics Software Architecture

Benjamin N. Lipchak
Thomas Frisinger
Karen L. Bircsak
Keith L. Comeford
Michael I. Rosenblum

The PowerStorm 4DT series of graphics devices established DIGITAL as the OpenGL performance leader for mid-range workstations, both on the DIGITAL UNIX and the Windows NT operating systems. Achieving this level of success required combining the speed of the Alpha microprocessor with the development of an advanced graphics subsystem architecture focused on exceptional software performance. The PowerStorm 4DT series of graphics adapters uses a modified direct-rendering technology and the Alpha CPU to perform geometry and lighting calculations.

The PowerStorm 4D40T, 4D50T, and 4D60T mid-range graphics adapters from DIGITAL have exceeded the performance of all OpenGL graphics devices costing as much as $25,000. In addition, these products achieved twice the price/performance ratio of competing systems at the time they were announced.

The PowerStorm 4DT series of mid-range graphics devices was developed in 1996 to replace the company's ZLX series. In its search for a vendor to replace the graphics hardware, DIGITAL found Intergraph Systems Corporation. This company had been designing three-dimensional (3-D) graphics boards for a few years and was then on its second-generation chip design. The schedule, cost, and performance of Intergraph's new design matched our project's target goals. Intergraph was building software for the Windows NT operating system on its Intel processor-based workstations, but was not doing any work for the UNIX operating system or the Alpha platform.

The goals of the PowerStorm 4DT project were to develop a mid-range graphics product powered by the Alpha microprocessor that would lead the industry in performance and price/performance.

This paper describes the competitive environment in the graphics industry at the conception of the PowerStorm 4DT project. It discusses our design decisions concerning the graphics subsystem architecture and performance strategy. The paper concludes with a performance summary and comparison in the industry.

Competitive Analysis

Overall performance of today's mid-range workstations is markedly better than that of just two years ago. This improvement is largely due to the dramatic increases in CPU speeds, both in the number of instructions executed per clock cycle and the number of clock cycles per second. Without trivializing the efforts of the CPU architects, such year-over-year increases in CPU performance have become the trend of the last decade, especially with the Alpha microprocessor.

More astounding is the central role that the graphics component of the workstation is playing in defining the overall performance of the workstation. We are in the age of visual computing. Whether or not an application requires 3-D graphics, even the most primitive applications often rely on a graphical user interface (GUI). As such, the graphical components of today's system-level benchmarks now carry significant weight.

More importantly, a prospective buyer often looks at results from standard graphics benchmarks as a general indication of a machine's overall performance. In the computer-aided design/computer-aided manufacturing (CAD/CAM) market, a customer typically buys a workstation to run a set of applications that has a large 3-D component. Performance is measured by how fast a workstation can create and manipulate 3-D objects. For the most part, this performance is determined wholly by the graphics subsystem. The hardware components of the graphics subsystem, however, vary from vendor to vendor and may or may not include the CPU.

Performance Metrics

Simply stated, the primary goal of the PowerStorm 4DT graphics device series was to provide the fastest mid-range OpenGL graphics performance while offering the best price/performance ratio. OpenGL is the industry-standard 3-D graphics application programming interface (API) and associated library that provides a platform-independent interface for rendering 3-D graphics.[1]

Quantifying performance can be an elusive goal. Product managers in our Workstation Graphics Group chose two metrics to compare the performance of the PowerStorm 4DT adapter to our competitors' products. The first metric was performance on the industry-standard OpenGL Viewperf benchmark, Conceptual Design and Rendering Software (CDRS).[2] This benchmark was chosen for its universal acceptance in the CAD/CAM and process control markets. When buyers compare graphics performance of two systems running OpenGL, the Viewperf scores are among the first measurements they seek. The second measurement was performance on the Pro/ENGINEER application from Parametric Technology Corporation (PTC).

The CDRS benchmark, as shown in Figure 1, was established by the OpenGL Performance Characterization (OPC) organization as one of several Viewperf viewsets. It emulates the variety of operations a user typically executes when running a CAD/CAM application. Specifically, this benchmark uses a series of tests that rotate a 3-D model on the screen in a variety of modes, including wireframe vectors, smooth-shaded facets, texturing, and transparency. Performance is measured by how many frames per second can be generated. Higher frame rates equate to faster and smoother rotations of the model. Each test carries a weight determined to roughly correspond to how important that operation is in a real-world CAD/CAM package. The test results are geometrically averaged to produce a composite score. This single number is a representation of the graphics performance of any given system.

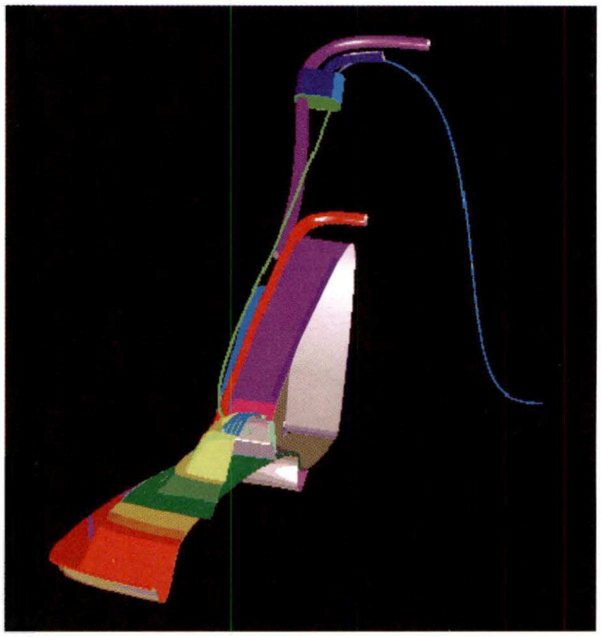
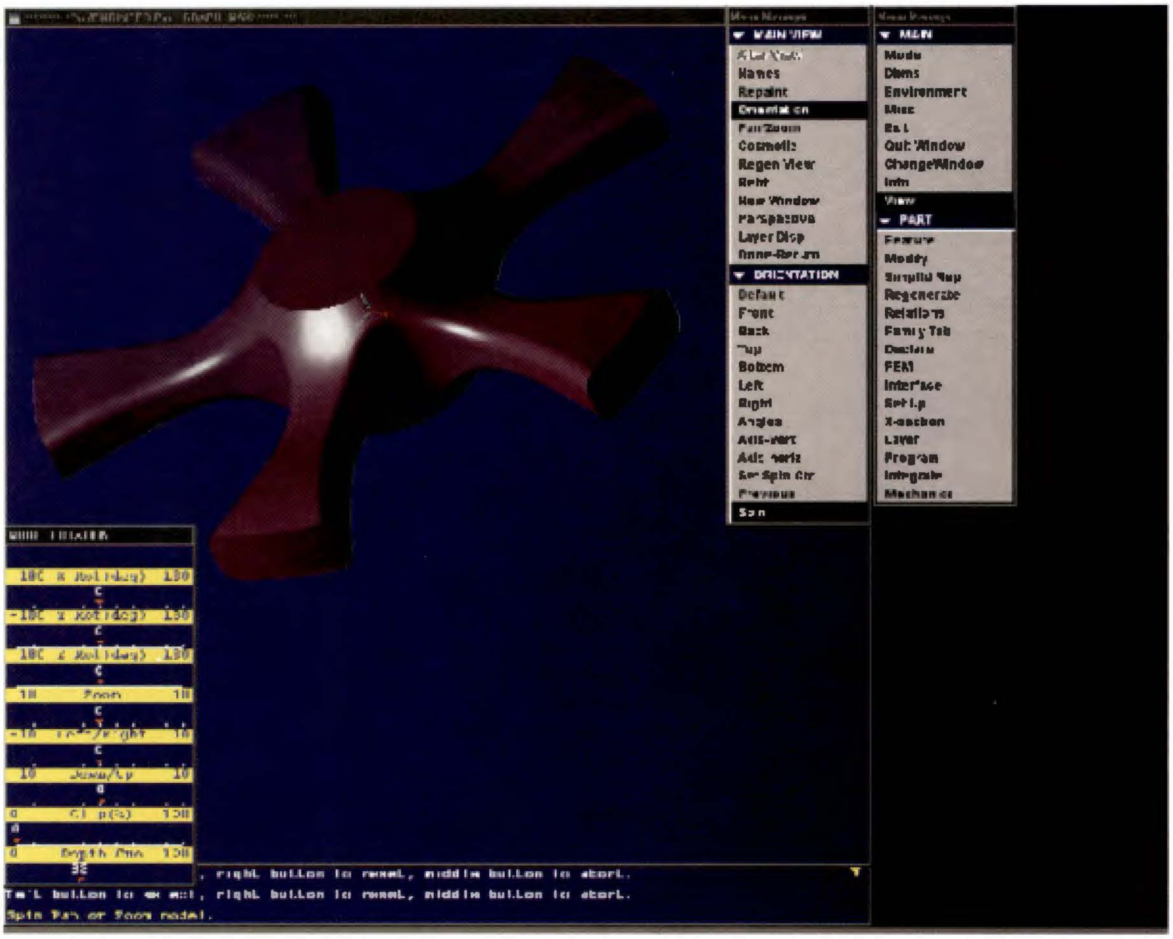


Figure 1
CDRS Viewperf Benchmark of OpenGL Performance

Although standard benchmarks are good performance indicators, they cannot substitute for actual performance on an application. To ensure that the PowerStorm 4DT adapter realized exceptional real-world performance, the second metric chosen was the CAD/CAM industry's market share leader, the Pro/ENGINEER application. PTC provides the industry with a set of playback files called trail files. As shown in Figure 2, each file contains a recording of a session in which a user has created and rotated a 3-D part. The recordings typically have large wireframe and smooth-shading components and little or no texture mapping. Performance is measured by how quickly a system can play back a trail file. The CDRS benchmark stresses only the graphics subsystem, but the Pro/ENGINEER trail file stresses the CPU and the memory subsystem as well.

Graphics Hardware Standards

In 1996, Silicon Graphics Inc. (SGI) captured the mid-range graphics workstation market with its Indigo2 Maximum IMPACT graphics subsystem powered by the MIPS R10000 microprocessor. DIGITAL, Sun Microsystems, and International Business Machines (IBM) Corporation had yet to produce a product with the performance SGI offered; instead, they competed in the low to lower mid-range graphics arena.



, right button to reset, middle button to abort.
Left button to exit, right button to reset, middle button to abort.
Spin Pan or Zoom model.

Figure 2
Screen Capture from the Pro/ENGINEER Trail File Used to Stress the PowerStorm 4DT Series

Hewlett-Packard was notably absent from either bracket due to its lack of a mid-range workstation with OpenGL graphics capability. Mid-range workstations can be loosely classified as costing from $15,000 to $40,000. Graphics performance in this price range differs, sometimes dramatically, from vendor to vendor.

Considering only raw graphics hardware performance, a vendor had to offer a certain level of performance to be competitive with SGI. By 1996 standards, a competitive device needed to be capable of achieving the following:

- 1 million Gouraud-shaded, 25-pixel, Z-buffered triangles per second
- 2 million flat-shaded, antialiased, 10-pixel vectors per second
- Trilinear, mipmapped, texture fill rates of 30 megapixels per second
- 24-bit deep color buffer
- 4-bit overlay buffer
- 4-MB dedicated or unified texture memory
- Dedicated hardware support for double buffering and Z-buffering
- Screen resolution of 1,280 by 1,024 pixels at 72 hertz

In 1996, the PowerStorm 4D60T, the most advanced graphics adapter in the new series, was capable of the following:

- 1.1 million Gouraud-shaded, 25- to 50-pixel, Z-buffered triangles per second
- 2.5 million flat-shaded, antialiased, 10-pixel vectors per second
- Trilinear, mipmapped, texture fill rates of greater than 30 megapixels per second
- 32-bit deep color buffer
- 8-bit overlay buffer
- 0- to 64-MB dedicated texture memory
- Dedicated hardware support for double buffering (including overlay planes) and Z-buffering
- Screen resolution up to 1,600 by 1,200 pixels at 76 hertz

It is important to understand that these are hardware maximums. The interesting work is not in achieving these rates under the best of conditions, but in achieving these rates under most conditions. To reiterate, building hardware that can theoretically perform well and building a system that performs well in benchmark applications are two distinctly different goals. The latter requires the former, but the former in no way guarantees the latter.

Different viewpoints on the best way to provide the highest level of performance have divided the industry into several camps. Workstation vendors must decide which approach best exploits the competitive advantages of their systems. In the mid-range workstation market, our graphics philosophy is decidedly different from that of our competitors. For the most part, DIGITAL is alone in its choice of a CPU-based, direct-rendering graphics architecture.

In the next section, we discuss the various graphics design architectures in the industry, focusing on the design of the PowerStorm series and comparing it with SGI's approach.

Graphics Subsystem Architectures

The two essential choices for graphics subsystem design are deciding between indirect and direct rendering and choosing whether the CPU or an application-specific integrated circuit (ASIC) performs the geometry and lighting calculations. In this section, we discuss the advantages and disadvantages of both rendering schemes and calculation devices and explore designers' decisions for graphics subsystem architectures.

By order of occurrence, 3-D graphics can be divided into three stages: (1) transferal of OpenGL API calls to the rendering library, (2) geometry and lighting, and (3) rasterization. In the next section, we compare direct and indirect image rendering.

Direct Versus Indirect Rendering

Before the popularization of the Windows NT operating system and the personal computer, almost all graphics workstations used the X Window System or a closely related derivative. The typical X Window System implementation is a standard client-server model.[3] An application that draws to the screen requests the X server to manage the graphics hardware on its behalf.

The graphics API, either Xlib for two-dimensional (2-D) applications or OpenGL for 3-D, was the functional breaking point. Traditionally, client applications would make graphics API calls to do drawing or another graphics-related operation. These calls would be encoded and buffered on the client side. At some point, either explicitly by the client or implicitly by the API library, the encoded and buffered requests would be flushed to the X server. These commands would then be sent to the X server over a transport such as the Transmission Control Protocol/Internet Protocol (TCP/IP), a local UNIX domain socket, or local shared memory.

When the requests arrived at the X server, it would decode and execute them in order. Many requests would then require the generation of commands to be sent to the hardware. This client-server model was named indirect rendering because of the indirect way in which clients interacted with the graphics hardware.

Direct rendering is a newer method often employed in the design of high-end graphics systems.[4,5] In this scheme, the client OpenGL library is responsible for all or most 3-D rendering. Instead of sending commands to the X server, the client itself processes the commands. The client also generates hardware command buffers and often communicates directly with the graphics hardware. In this rendering scheme, the X server's role is greatly diminished for 3-D OpenGL requests but remains the same for 2-D Xlib requests.

The designers chose to support direct rendering for the PowerStorm 4DT adapter. Direct rendering offers considerably better performance than indirect rendering. Note, however, direct rendering does not preclude indirect rendering. All devices that support direct rendering under the X Window System also support indirect rendering.

In the following subsections, we discuss the advantages and disadvantages of direct and indirect rendering. We also explain the impetus for making the PowerStorm 4DT adapter the first graphics device from DIGITAL capable of direct rendering.

Indirect Rendering One advantage of indirect rendering that should never be underestimated is its proven track record. This technology is widely accepted and understood. It offers network transparency, which means a client and server need not reside on the same machine. A client can redirect its graphics to any machine running an X server as long as the two machines are connected on a TCP/IP network. This model worked well until faster CPUs and graphics devices were developed. The protocol encode, copy, and decode overhead associated with sending requests to the server became a bottleneck.

The increased use of display lists provided an intermediate solution to this problem. Display lists are a group of OpenGL commands that can be sent to the X server once and executed multiple times by referencing the display list ID instead of sending all the data each time. Display lists dramatically reduced communication overhead and returned graphics to the point at which communication to the X server was no longer the bottleneck.

Unfortunately, display lists had significant disadvantages. Once defined, they could not be modified. To achieve performance using indirect rendering, almost all OpenGL commands had to be collected into display lists. This caused resource problems because display lists could be quite large and had to be stored in the X server until explicitly deleted by the client. Probably the greatest disadvantage was that display lists were generally awkward for application programs to use. Application programmers prefer the more straightforward method of immediate-mode programming by which commands are called individually. For these reasons, indirect rendering proved to be insufficient, even with the advent of display lists.

Direct Rendering The PowerStorm 4DT project team was committed to designing a product with leadership performance for both the display-list-mode and immediate-mode rendering. The designers realized early that they would have to adopt direct rendering to address the performance problems with immediate-mode indirect rendering.

As mentioned earlier, the philosophy behind classical direct rendering is that each client handles all OpenGL processing, creates a buffer of hardware commands for the device, and then sends the commands to the device without any X server interaction. This model has several drawbacks. First, access to the graphics hardware is difficult to synchronize between clients and the X server. Second, windows and their properties such as position and size have to be maintained by the clients, which also requires a complex synchronization design. SGI used this model for its IMPACT series of graphics devices.

The PowerStorm 4DT designers took a more conservative approach, based largely on the same model. One fundamental difference is that each client generates hardware command buffers in shared memory. The client then sends requests to the X server telling it where to locate the hardware commands. The X server sets up the hardware to deal with window position and size and then initiates a direct memory access (DMA) of the hardware command buffer to the graphics device. Essentially, the X server becomes an arbitrator of hardware buffers. This approach worked quite well, because the X server was the logical place for synchronization to occur and it already maintained window properties. We were able to have all the performance benefits of classical direct rendering without the pitfalls.

One implication of direct rendering is that the client and the server have to be on the same physical machine. When first evaluating direct rendering, designers were curious to determine how often our customers used this configuration; that is, did most users perform their work and display their graphics on the same computer? Our surveys showed that more than 95 percent of users did display their graphics locally. The remaining 5 percent rarely cared about performance. Today, this may seem obvious; two years ago, it could not be assumed.

Direct rendering offered a huge performance improvement to nearly all our customers. The performance gains were two to four times the performance of indirect rendering.

Direct-rendering 2-D Most graphics device implementations use direct rendering only for OpenGL, because indirect rendering of immediate-mode OpenGL is protocol rich. As mentioned previously, the transferal of this protocol to the X server can be quite expensive. One interesting aspect of our design is its support for direct rendering of 2-D Xlib calls.

Other graphics vendors consider 2-D performance important only for 2-D benchmarks. These benchmarks, which largely stress the graphics hardware's ability to draw 2-D primitives quickly, can generate a lot of work for the hardware with relatively few requests. Unlike 3-D, these requests do not need much geometry processing before they can be sent to the hardware. This means that very little protocol is needed to saturate the hardware. As long as the protocol generation does not produce a bottleneck, indirect rendering performs as well as direct rendering. In addition, given that OpenGL benchmarks like CDRS have almost no 2-D component, it seems reasonable to conclude that indirect-rendered 2-D should suffice.

Benchmarks often are not sufficiently representative of real applications, especially when they isolate 2-D and 3-D operations. CAD/CAM applications typically have a substantial 2-D GUI, which interacts closely with the 3-D components of the application. A benchmark that exercises both 2-D and 3-D by emulating a user session on an application will provide results that more accurately reflect the performance witnessed by an end user. These benchmarks simply measure how long it takes to complete a session, so both 3-D and 2-D performance impact the overall score.

Our research showed that with a highly optimized OpenGL implementation, in many cases it was no longer the 3-D components that slowed down a benchmark, but the 2-D components. Further examination revealed that it was the same protocol bottleneck evident with indirect-rendered OpenGL. Applications were generating relatively small drawing operations with many drawing attribute changes intermixed, such as draw line, change color, draw line, change color, and so forth. This type of request stream tends to generate tremendous amounts of protocol, unlike 2-D benchmarks that rarely change drawing attributes.

Accordingly, 2-D direct rendering presented itself as the logical solution. With the direct-rendering infrastructure and design already in place, developers simply needed to extend it for 2-D/Xlib. This required the development of two additional libraries: the Vectored X library and the Direct X library (unrelated to Microsoft's DirectX API).

The Vectored X library replaced the preexisting Xlib. It allows devices that support direct rendering to vector, or redirect, Xlib function calls to direct-rendering routines instead of generating the X protocol and sending it to the X server. If a graphics device does not support direct rendering, it defaults to the generic protocol-generating routines. It is important to understand that this is a device-independent library responsible only for vectoring Xlib calls to the appropriate library.

The Direct X library, on the other hand, is a device-dependent library. It contains all the vectored functions that the Vectored X library calls when the device supports direct rendering. This library operates in much the same way as the direct-rendering OpenGL library. It processes the requests and places graphics hardware commands in a shared memory buffer. The X server later sends the buffer to the graphics device by DMA.

The entire functionality of the X library is not implemented through direct rendering for several reasons. In many cases, a shared resource resides in the server (e.g., the X server performs all pixmap rendering). In other cases, the hardware is not directly addressable by the client (e.g., the X server handles all frame buffer reads). Often the client does not have access to all window information that the server maintains (e.g., the X server handles all window-to-window copies). Fortunately, these operations are either not frequently used, not expected to be fast, or easily saturate the hardware.

Further details of the Vectored X library and Direct X library are beyond the scope of this paper. The concept of direct-rendered 2-D, however, is sound. It has helped DIGITAL outperform other vendors on many application benchmarks that were largely focused on OpenGL but had significant 2-D components. Our 2-D direct-rendering technology has also enhanced 2-D performance and response time for the many thousands of exclusively 2-D applications for the X Window System.

Geometry and Lighting

The geometry and lighting phase can be performed by the host CPU or by a specialized, high-speed ASIC, which is typically located on the graphics device. Regardless of where these calculations take place, the general idea is that the user's vertices are transformed and lit, then fed to the rasterizer. Since the rasterizer is on the graphics device, choosing the host to do the geometry and lighting implies that the transformed and lit vertices are then sent across the bus to the rasterizer. The use of a specialized ASIC implies that the user's vertices are sent across the bus, transformed and lit by the custom ASIC, and then fed directly to the rasterizer. The information transferred across the bus is obviously different, but in terms of amount of data per vertex, it is approximately the same. Therefore, bus bandwidth does not become a deciding factor for either design.

Host CPU Geometry and Lighting Traditionally, DIGITAL has chosen the host CPU to perform the geometry and lighting calculations. The PowerStorm project designers chose this approach because of the Alpha microprocessor's exceptional floating-point speed, and because almost all 3-D calculations involve floating-point values. At the time this project was conceived, the only general-purpose, widely available processor capable of feeding more than 1 million transformed and lit vertices per second to the hardware was the Alpha CPU. An additional benefit of having the Alpha CPU do the work was an overall cost reduction of the graphics device. Custom ASICs are expensive to develop and manufacture.

Another important and related advantage is that our software becomes proportionally faster as clock speeds rise on available Alpha microprocessors. This results in a near linear performance increase without any additional engineering cost. For example, using the same software, a 500-megahertz (MHz) Alpha microprocessor is able to produce 25 percent more vertices per second than a 400-MHz Alpha microprocessor. Because of this, developers can write optimized Alpha code once and reuse it for successive generations of Alpha CPUs, reaping performance improvements with virtually no further invested effort.

It is obvious that rendering can proceed no faster than vertices can be generated. If the OpenGL library can transform and light only 750,000 vertices per second, and the graphics device can rasterize 1 million, the effective rendering rate will be 750,000. In this example, the OpenGL geometry and lighting software stages are the bottleneck. However, if the numbers were reversed, and the hardware could only rasterize 750,000 vertices while the OpenGL software provided 1 million, the rasterization hardware would become the bottleneck.

Thus far, we have discussed two potential bottlenecks: the OpenGL implementation itself and the rasterization hardware. The third and potentially most damaging bottleneck may be the client's ability to feed vertices to the OpenGL library. It should be clear that this is the top level of vertex processing. The OpenGL library can render no faster than the rate at which the client application feeds it vertices. Consequently, the rasterizer can render primitives no faster than the OpenGL library can produce them. Thus, a bottleneck in generating vertices for the OpenGL library will slow the entire pipeline. Ideally, we would like each level to be able to produce at least as many vertices as the lower levels can consume.

Clearly, the performance of the application, in terms of handing vertices to the OpenGL library, is a function of CPU speed. This is only an issue for applications that have large computation overhead before rendering. Currently, almost all graphics benchmarks have little or no computation overhead in getting vertices to the OpenGL library. Most attributes are precomputed, since they are trying to measure only the graphics performance and throughput. For the most part, this holds true for the traditional CAD/CAM packages. However, some emerging scientific visualization applications as well as some high-end CAD applications require significant compute cycles to generate the vertices sent to the OpenGL library. For these applications, only the DIGITAL Alpha CPU-based workstations can produce the vertices fast enough for interactive rates.

There are some potential disadvantages to this design. Namely, the CPU is responsible for both the application's and the graphics library's computations. If the application and the OpenGL implementation must contend for compute cycles, overall performance will suffer. Analysis of applications revealed that typical 3-D and 2-D graphics applications do internal calculations followed by rendering. Only under rare circumstances do the two processes mix with a substantial ratio. If the applications should start mixing their own processing needs with those of the OpenGL library, the notion of host-based geometry would need to be revisited.

Another potential disadvantage is the rate at which Alpha CPU performance increases versus the rate at which the rasterizer chip's performance increases. The emerging generation of graphics devices is capable of rasterizing more than 4 million triangles per second. It is unknown whether future generations of the Alpha CPU will be able to feed the faster graphics hardware.

ASIC-based Geometry and Lighting Performing geometry and lighting calculations with a custom ASIC on the graphics device is often referred to as OpenGL in hardware because most of the OpenGL pipeline resides in the ASIC. The OpenGL library is limited to handing the API calls to the hardware. SGI has adopted the ASIC-based approach for many generations of workstations and graphics devices. In this section, we discuss why this method works for them and its potential shortcomings.

SGI workstations use either the R4400 or the R10000 CPU developed by MIPS Technologies. Although these CPUs have good integer performance, their floating-point performance cannot generate the number of vertices that the Alpha CPU can. As a consequence, SGI has to use the custom-graphics ASIC approach. One advantage to the custom ASIC is the decoupling of graphics from the CPU. Since each can operate asynchronously, the application has full use of the CPU.

Typically, custom geometry ASICs, also known as geometry engines, perform better than a general-purpose CPU for several reasons. First, the custom ASIC must perform only a well-understood and limited set of calculations. This allows the ASIC designers to optimize their chip for these specific calculations, releasing them from the burden and complexity of general-purpose CPU design.

Second, the graphics engine and the rasterizer can be tightly coupled; in fact, they can be located on the same chip. This allows for better pipelining and reduced communication latencies between the two components. Even if the geometry engine and rasterizer are located on different chips, which is not at all uncommon, a much stronger coupling exists between the geometry engine and the rasterizer than does between the host CPU and rasterizer.

Third, geometry engines can yield high performance when executing certain display lists. The use of a display list allows an object to be quickly re-rendered from a different view by changing the orientation of the viewer and reexecuting the stored geometry. If the display list can fit within the geometry engine's cache, it can be executed locally without having to resend the display list across the bus for each execution. This helps alleviate the transportation overhead in getting the display list data over the bus to the graphics device. It is unclear how often this really happens since rasterization is typically the bottleneck. If the display list is filled with many small area primitives, however, its use can result in noticeable performance gains. Geometry engines often have a limited amount of cache. If an application's display list exceeds the amount of cache memory, performance degrades significantly, often to below the performance attainable without a geometry accelerator. Our research shows that display list sizes used by applications increase every year; therefore, cache size must increase at the same rate to maintain display list performance advantages.

The primary disadvantage of using custom ASICs to perform the geometry and lighting calculations is the expense associated with their design and manufacture. In addition, a certain risk is involved with their development: hardware bugs can seriously impact a product's viability. Fixing the bugs causes the schedule to slip and the cost to rise. Hardware bugs discovered by customers can be devastating. With host-based geometry, a software fix in the OpenGL library can easily be incorporated and distributed to customers.

A sometimes unrecognized disadvantage of dedicated geometry engines is that they are bound to fixed clock rates, with little room for scalability. Although this is true of most CPU designs, CPU vendors can justify the engineering effort required to move to a faster technology, because of competitive pressures and the larger volume of host CPU chips.

Rasterization

During the rasterization phase, primitives are shaded, blended, textured, and Z-buffered. In the early years of raster-based computer graphics, rasterization was done using software. As computer graphics became more prevalent, graphics performance became an issue. Because rasterization is highly computational and requires many accesses to frame buffer memory, it quickly became the performance bottleneck. Specialized hardware was needed to accelerate the rasterization part of graphics. Fortunately, hardware acceleration of rasterization is well understood and is now the de facto standard. Today, nearly every graphics device has rasterization hardware. Even low-priced commodity products have advanced raster capabilities such as texture mapping and antialiasing.

In the next section, we relate our strategy for obtaining optimal graphics software performance from an Alpha processor-based system.

Performance Strategy

The goals of the PowerStorm 4DT program were largely oriented toward performance. Our strategy consisted of having a generic code path and then tuning performance where necessary using Alpha assembly and integrated C code.

Performance Architecture

The designers optimized the software performance of the PowerStorm 4DT series within the framework of a flexible performance architecture. This architecture provided complete functionality throughout the performance-tuning process, as well as the flexibility to enhance the performance of selected, performance-sensitive code paths.

In this context, code paths refer to the vertex-handling routines that conduct each vertex through the geometry, lighting, and output stages. Whereas most OpenGL API calls simply modify state conditions, these vertex routines perform the majority of computation. This makes them the most likely choices for optimization.

The Generic Path A solid, all-purpose code base written in C and named the generic path offers full coverage of all OpenGL code paths. The generic path incurs a significant performance penalty because its universal capabilities require that it test for and handle every possible combination of state conditions. In fact, under certain conditions, the generic path is incapable of driving the hardware at greater than 33 percent of its maximum rendering rate. The generic path assumes responsibility for the rare circumstances that are not deemed performance-sensitive and thus not worthy of optimization. It also acts as a safety net when high-performance paths realize mid-stride that they are not equipped to handle new, unanticipated conditions.

Multicompiled Speed of Light (SOL) Paths High-performance SOL paths provide greatly increased performance where such performance is necessary. Under prescribed conditions, SOL paths replace the generic path, yielding equivalent functionality with performance many times that of the generic path. SOL paths were written for the combinations of state conditions exercised most frequently by the target applications and benchmarks.

The developers responsible for performance tuning designed two classes of SOL paths. First, they generated a large number of SOL paths by compiling a C code template multiple times. Whereas the generic path is composed of several routines, each corresponding to a single stage of the pipeline, a multicompiled SOL path integrates these stages into a monolithic routine. Each compilation turns on and off a different subset of state conditions, resulting in integrated paths for every combination of the available conditions. This multicompilation of integrated SOL paths yields the following benefits:

- The C compiler is allowed a broader overview of the code and can more wisely schedule instructions. In contrast, the generic path is composed of several individual stages. These relatively short routines do not provide the C compiler with enough space or enough scope to make informed and effective, instruction-ordering decisions. Multicompiling the various stages into a series of monolithic, integrated routines relieves each of these problems.
- The multicompilation assumes a fixed set of conditions for each generated path. This eliminates the need for run-time testing of these conditions during each execution of the path. Instead, such testing is necessary only when state conditions change. Validation, as this testing is called, determines which new path to employ, based on the new state conditions. With the great number and complexity of state conditions influencing this decision, validation can be an expensive process. Performing validation only in response to state changes, rather than for every vertex, results in significant performance gains.
- The SOL path coverage at least doubles every time that support for a new state condition is added to the template. Each new condition increases the number of combinations of conditions being multicompiled into SOL paths by a factor of two or more. An adverse side effect of this strategy is that the compile time and resulting library size will increase at the same rate as the SOL path coverage.

Assembly Language SOL Paths Hand-coded Alpha assembly language paths constitute the other class of high-performance SOL paths. These paths, designed specifically for extremely performance-sensitive conditions, require much more time and attention to produce. Taking advantage of the many features of the Alpha microprocessor transforms assembly language coding from a science into an art form.[6] The Alpha assembly coders kept the following issues foremost in their minds:

- The 21164 and subsequent Alpha microprocessors are capable of quad-issuing instructions, which means that as many as four instructions can be initiated during each cycle. The combination of instructions that may be issued, however, depends on the computational pipelines and other resources employed by each instruction. Coders must carefully order instructions to gain the maximum benefit from the multiple-issue capability.
- As a consequence of the above restrictions, integer and floating-point operations must be scheduled in parallel. With few exceptions, only two floating-point and two integer instructions can be issued per cycle. Efficiency in this case requires not only local instruction-order tweaking but also global changes at the algorithmic level. Integer and floating-point operations must be balanced throughout each assembly routine. If a particular computation can be easily performed using either integer math or floating-point math, the choice is made according to which pipeline has more free cycles to use.
- Register supply is another factor that affects the design of an assembly language routine. Although the Alpha microprocessor has a generous number of registers (32 integer and 32 floating-point), they are still considered a scarce resource. The coder must organize the routine such that some calculations complete early, freeing registers for reuse by subsequent calculations.
- The crucial performance aspect of assembly coding is transporting the data where and when it is needed. The latency of loading data from main memory or even from cache into a register can easily become any routine's bottleneck. To minimize such latencies, load instructions must be issued well in advance of a register's use; otherwise, the pipeline will stall until the data is available. In an ideal architecture with an infinite quantity of registers, all loads could be performed well in advance. Unfortunately, due to the scarce amount of free registers, the number of cycles available between loading a register and its use is frequently limited.

Each of these assembly language programming considerations requires intense attention but yields unmatched performance.

Performance Tuning

After reviewing benchmark comparisons and recommendations from independent software vendors, we determined which areas required performance improvement. We approached performance tuning from two directions: either by increasing SOL path coverage or improving the existing SOL code.

Increasing SOL path coverage was the more straightforward but the more time-consuming approach. If an SOL path did not exist for a specific condition, a new one would have to be written. Adding a new option to the multicompilation template required a significant effort in some cases. Implementing a new assembly language SOL path always required significant effort.

Improving the performance of an existing SOL path required an iterative process of profiling and recoding. We employed the DIGITAL Continuous Profiling Infrastructure (DCPI) tools to analyze and profile the performance of our code.[7] DCPI indicated where bottlenecks occurred and whether they were due to data cache misses, instruction slotting, or branch misprediction. This information provided the basis for obtaining the maximum performance from every line of code.

Development of 3-D Graphics on Windows NT

At the start of the PowerStorm 4DT project, the Windows NT operating system was an emerging technology. The DIGITAL UNIX platform held the larger workstation market share, while Windows NT accounted for only a small percentage of customers. For that reason, designers targeted performance for applications running on DIGITAL UNIX and developed 3-D code entirely under that operating system.

Nevertheless, we recognized the potential gains of developing 3-D graphics for the Windows NT system. One of the company's goals was to be among the first vendors to provide accelerated OpenGL hardware and software for Windows NT.

With a concerted effort and a few compromises, the team developed the PowerStorm 4DT into the fastest OpenGL device for Windows NT, a title that was held for more than 18 months. To achieve this capability, the designers made the following key decisions:

- To write code that was portable between the DIGITAL UNIX and Windows NT systems.
- To dedicate two people to the integration of the DIGITAL UNIX-based code into the Windows NT environment. Most OpenGL code was operating-system independent, but supporting infrastructure needed to be developed for Windows NT.
- To use Intergraph's preexisting 2-D code and to avoid writing our own. Intergraph provided us with a stable 2-D code base for Windows NT. This code base had room for optimization, but further optimization of the 3-D code took precedence.
- To ship the graphics drivers for DIGITAL UNIX first, and the graphics drivers for Windows NT three months later. In this way, we allowed the DIGITAL UNIX development phase to advance unimpeded by the efforts to port Windows NT.

Results and Conclusion

In August of 1996, the PowerStorm 4D60T graphics adapter was best in its price category with a CDRS performance number of 49.01 using a 500-MHz Alpha processor. It yielded a new price/performance record of $321 per frame per second. At the same time, SGI attained a CDRS number of only 48.63 on a system costing nearly three times as much.

Figure 3 shows the relative performance of the PowerStorm 4D60T for four of the major Viewperf benchmarks. The viewsets are based on the following applications: CDRS, a computer-aided industrial design package from PTC; Data Explorer (DX), a scientific visualization package from IBM; DesignReview (DRV), a model review package from Intergraph; Advanced Visualizer, a 3-D animation system from Alias/Wavefront (AWadvs).

The PowerStorm 4D60T mid-range graphics adapter easily outperformed the Indigo2 High IMPACT system from SGI by a wide margin and even surpassed SGI's more expensive graphics card, the Indigo2 Maximum IMPACT, by a factor of more than 2:1 in price/performance on these benchmarks. Figure 4 shows that the PowerStorm 4D60T was the performance leader in three of the four benchmarks. SGI has yet to produce a graphics product in this price range that outperforms the PowerStorm 4D60T.

Acknowledgments

The authors would like to acknowledge the many other engineers who made the PowerStorm 4DT project a successful one, including Monty Brandenburg, Shih-Tang Cheng, Bill Clifford, John Ford, Chris Kniker, Jim Rees, Shuhua Shen, Shree Sridharan,

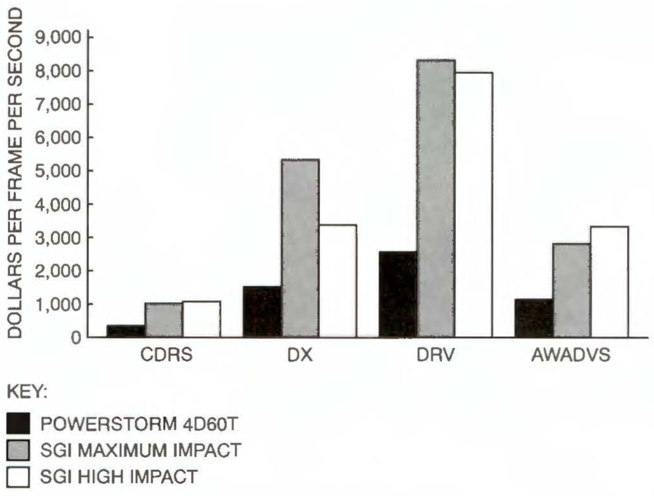


Figure 3
Price/Performance Comparison of Graphics Adapters on Viewperf Benchmarks

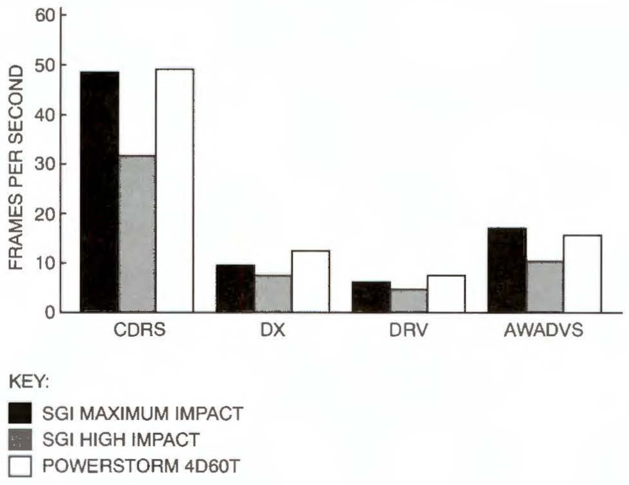


Figure 4
Performance of Graphics Adapters on Viewperf Benchmarks

Bruce Stockwell, and Mark Yeager. We would also like to thank the Graphics Quality Assurance Group and the Workstation Application Benchmarking Group for their unending patience and cooperation.

References

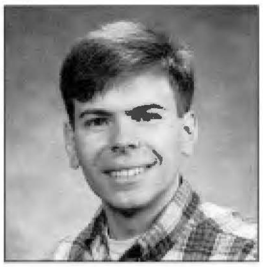
1. M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification* (Mountain View, Calif.: Silicon Graphics, Inc., 1995).
2. The OpenGL Performance Characterization Project, http://www.specbench.org/gpc/opc.static.
3. R. Scheifler and J. Gettys, *X Window System* (Boston: Digital Press, 1992).
4. H. Gajewska, M. Manasse, and J. McCormack, "Why X Is Not Our Ideal Window System," *Software Practice and Experience* (October 1990).
5. M. Kilgard, "D11: A High-Performance, Protocol-Optional, Transport-Optional Window System with X11 Compatibility and Semantics," *The Ninth Annual X Technical Conference,* Boston, Mass. (1995).
6. R. Sites and R. Witek, *Alpha AXP Architecture Reference Manual* (Boston: Digital Press, 1995).
7. J. Anderson et al., "Continuous Profiling: Where Have All the Cycles Gone?" *The 16th ACM Symposium on Operating Systems Principles,* St. Malo, France (1997): 1–14.

General References

J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics Principles and Practice* (Reading, Mass.: Addison-Wesley, 1993).

M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide* (Reading, Mass.: Addison-Wesley, 1997).

Biographies



Benjamin N. Lipchak
Benjamin Lipchak joined DIGITAL in 1995 to develop software for the PowerStorm 4DT graphics adapter and later developed 3-D software for the PowerStorm 4D30T project. A senior software engineer in the Workstation Graphics Group, he is currently leading the software effort of a new graphics project. Benjamin received B.S. (highest honors) and M.S. degrees in computer science from Worcester Polytechnic Institute. He is the recipient of the Salisbury Award in Computer Science.

Thomas Frisinger
Tom Frisinger was a senior software engineer in the Workstation Graphics Group at DIGITAL for three years. During that time, he contributed to nearly all aspects of the PowerStorm 4DT project. As a member of the core software engineering team, he helped develop software for the 4D40T, 4D50T, and 4D60T graphics adapters as well as the 4D30T and 4D51T models. He was also part of the core software design team for the 4D31T graphics accelerator. Tom is currently doing research and development in PC graphics for ATI Research, Inc.

Karen L. Bircsak
As one of the developers of the PowerStorm 4DT graphics adapter, Karen Bircsak designed and implemented enhancements to the X library and contributed to other software development areas. A principal software engineer in the Workstations Graphics Group, Karen is currently working on supporting new graphics hardware. Prior to joining DIGITAL in 1995, she held software engineering positions at Concurrent Computer Corporation and Data General Corporation. She earned a B.S. in computer science and engineering from the University of Pennsylvania in 1984 and an M.S. in computer science from Boston University in 1990.

Keith L. Comeford
Keith Comeford is a principal software engineer in the Workstation Graphics Development Group. He is currently working on the next generation of graphics cards and accelerators for DIGITAL. Keith was the project leader for the Windows NT drivers for the PowerStorm 4D40T/50T/60T graphics cards. In previous project work, Keith contributed significantly to the GKS and PHIGS implementations in a variety of capacities from developer to project leader for more than10 years. Keith joined DIGITAL in 1983 after receiving a B.S. in computer science from Worcester Polytechnic Institute.

Michael I. Rosenblum
Mike Rosenblum is a consulting software engineer at DIGITAL and the technical director for the Workstations Business Segment Graphics Group. He was the project leader and architect of the PowerStorm 4DT series and implemented some of its 2-D DDX code. Currently, he is managing two graphics projects and consulting to the company on graphics-related issues. Mike joined DIGITAL in 1981, to work on the terminal driver in the VMS Engineering Group. Later he helped design the company's first workstations. He has a B.S. in computer science from Worcester Polytechnic Institute and is a member of the ACM.

Robert J. Walsh

DART: Fast Application-level Networking via Data-copy Avoidance

The goal of DART is to effectively deliver high-bandwidth performance to the application, without a change to the operating system call semantics. The DART project was started soon after the first DART switch was completed, and also soon after line-rate communication over DART was achieved. In looking forward to gigabit class networks as the next hurdle to conquer, we foresaw a need for an integrated hardware-software project that addressed fundamental memory bandwidth bottleneck issues through a system-level perspective.

The Ethernet supported large 100-node networks in 1976.[1] By 1985, 10 Mb/s Ethernet had been available for a while, even for PCs. However, high-performance hardware and software lagged, due to system bottlenecks above the physical layer. The premier implementations for UNIX were achieving only 800 kb/s (8 % of 10 Mb/s) in benchmark scenarios on common system platforms of the day.[2]

The deployment of 100 Mb/s fiber distributed data interface (FDDI) provided an order of magnitude bandwidth increase in the link speed around 1987. However, the end system could not saturate the link on generally available machines and operating systems until 1993,[3] when Transmission Control Protocol (TCP) improvements and a CPU capable of 400 million operations per second became available.[a] Once again, high-performance hardware and software lagged the potential provided by the physical layer.

The current technological approach is switching. Gigabit-class links and adapters, such as 622 Mb/s asynchronous transfer mode (ATM), are becoming available. Since ATM links are dedicated point-to-point connections, the use of 622 Mb/s in switch-to-switch links and at the periphery implies that one ought to be able to move data at gigabit rates.

Switched capacity promises a lot to servers; however, mainstream systems are not currently capable of effectively using the bandwidth. The DART project attempts to avoid the Ethernet and FDDI scenarios where end-system performance lags physical-layer potential.

One of the early goals was to go beyond simple benchmark scenarios where line rate communication connects a phony bit source to a phony bit sink, with the CPU saturated. The context for the work was to connect two applications at high speed, leaving CPU

[a]The TCP improvements included a small architectural update, the window scaling extension, to abstractly support the advertisement of more than 64 kbytes of receive buffering. The rest of the improvements derived from implementation efforts to increase the actual buffering allocated to advertised TCP windows, and to improve the segmentation of the TCP byte stream into packets.

resources available to execute the applications. In the past, the CPU had been saturated in Ethernet and FDDI quests for line rate communication.

Layering

The motivation for DART arises from the specific layering and abstraction used in BSD-derived UNIX systems, but the context is sufficiently general that the problem and solution have wide applicability. Since various layers within system software will be referenced repeatedly, we introduce them using Figure 1.

The *application* generates and consumes data. It tells the operating system which data to communicate when, by using read and write system calls.

The *socket layer* moves data between the operating system and the application. It also synchronizes the application with the networking protocols based on data and buffer availability.

The *transport protocol layer* provides a connection to the remote peer. In the case of TCP, the connection is a reliable byte stream. TCP takes on the responsibility of retransmitting lost or corrupted data, and of ignoring reception of retransmitted data that was previously received.

The *network protocol layer* provides an abstract address and path to the remote host. It hides the various hardware-specific addresses used by the various media in existence. In the case of IP, fragmentation allows messages to traverse media which have different frame sizes.

A conventional *driver layer* moves data between the network and the system. It uses buffers and data structures whose representation percolates throughout all the operating system networking layers.

The DART Concept

DART increases network throughput and decreases system overheads, while preserving current system call semantics. The core approach is data copy avoidance, to better utilize memory bandwidth.

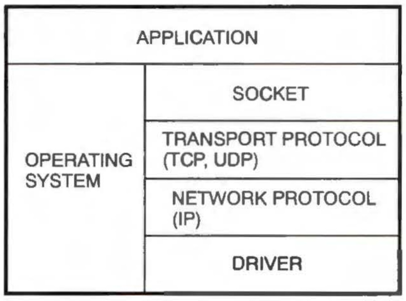


Figure 1
Software Layering

Memory bandwidth is a scarce resource that must not be squandered. In DIGITAL's transition from MIPS processor systems to Alpha processor systems, CPU performance increased more rapidly than main memory bandwidth. It took approximately 340 μs to move 4500 bytes on the MIPS-based DECstation 5000/200, and approximately 200 μs on the Alpha-based DEC 3000/500. In the same time, the fixed per-packet costs were reduced by a factor of three or more. (General trends are also stated in Reference 4.)

One breakdown of networking costs is reported in Reference 5. The variable per-byte costs reported there are all associated with memory bandwidth, which is improving slowly. The fixed per-packet costs in the driver, protocol, and operating system overhead are all generally associated with the CPU, which is improving rapidly. Thus, we focus on the per-byte memory bandwidth issues as those most needing architectural improvement.

A traditional system follows the networking subsystem model implemented within the BSD releases of UNIX, shown in Figure 2. An application uses the CPU to create data (1), the socket portion of the system call interface copies the data into operating system buffers (2 and 3), the network transport protocol checksums the data for error detection purposes (4), and the device driver uses programmed input/output (I/O) or direct memory access (DMA) to move the data to the network (5). Graphs showing the dominant costs of checksumming and kernel buffer copies are presented in Reference 6.

These five memory operations are a profligate waste of memory bandwidth. A system with a 300 Mbyte/s memory system would achieve at most 300*8/5 = 480 Mb/s I/O rates. The system would be saturated.

The DART model is shown in Figure 3. The DART model is that data is created (1) and sent (2). Two memory operations make efficient use of the memory bandwidth.

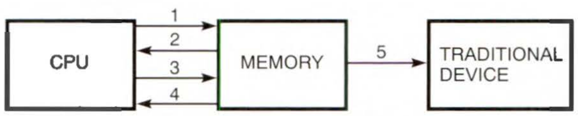


Figure 2
BSD Copy-based Architecture

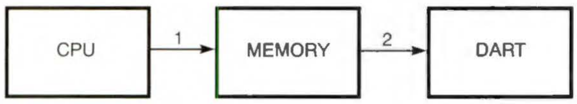


Figure 3
DART Zero-copy Architecture

Squandering of memory bandwidth is avoided. A system with a 300-Mbyte/s memory system would encounter the larger bound of 300*8/2 = 1200 Mb/s for I/O rates. Resources are available for the application even when running at line rate.[b]

To support the DART concept, we need a system perspective that integrates the hardware and software changes implied by the DART model. Hardware is responsible for checksumming instead of software. Hardware is solely responsible for data movement, instead of redundant actions by both hardware and software. These hardware changes are bounded and generic.

Operating system software retains the application interface and general coding of the BSD UNIX implementation. Extensive changes are unnecessary, since the focus is the core lines that represent data movement consumption of memory bandwidth. Extensive changes are also undesirable, since there is a large base of software written to the current properties of the BSD networking subsystem.

The DART Hardware

The first implementation of the DART concept is a high-performance 622-Mb/s ATM network adapter for the PCI bus called DART. DART's design reflects an awareness of the interactions of the components of the system in which it is placed. The PCI bus, main memory, cache, and system software can all be used efficiently.

Store-and-Forward Buffering and DMA

DART is an adapter that connects a gigabit-class network to a gigabit-class I/O bus, and is appropriate for systems with gigabit-class memory systems. DART is focused on the server market where a slight increase in adapter cost can be acceptable if the system performance is significantly improved, since main memory and other costs dominate the cost of the DART adapter.

DART alleviates main memory bottlenecks through a store-and-forward design, as shown in Figure 4. Traditional networking software subsystems and applications perform at least five memory operations to create, copy, checksum, and communicate data. DART's *exposed buffering* allows data to be created and communicated with just two main memory operations. The adapter memory is a resource that can be better utilized by exposing it to the operating system, and better performance results as well. This is similar to the exposure of the CPU-internal mechanism in the CISC-RISC (complex to reduced instruction set) transition.

DART contains a number of features to make the store-and-forward design effective. DART's bus master and receiver summarize network transport protocol checksums for software. DART's bus master provides byte-level scatter-gather data movement to support communication out of application buffers, not just operating system buffers. DART provides packet headers for software to parse so that software can direct the bus master to place received data in the application's buffers when the application desires, without operating system copy overhead.

Buffering Design An ATM segmentation and reassembly (SAR) chip accesses virtual circuit state for each cell, and operates on 48-byte cell payloads. The payload naturally corresponds to a burst-mode operation, leading to the use of synchronous dynamic DRAM (SDRAM) to buffer cells. The circuit state is generally smaller and randomly accessed, leading to the use of static RAM (SRAM) for control information. Dividing the data storage architecture into two parts allows the interface designs to be tailored to the characteristics of the data type in question.

The DART prototype uses 16 Mbytes of SDRAM for the data memory. The prototype uses 1 Mbyte of SRAM for the control memory. The SDRAM supports hardware-generated transmissions, aggregation of data for efficient PCI and host memory interactions;[c] and buffering for received data until the application indicates the proper destination for it. The SRAM contains the SAR intermediate state; with a large number of virtual circuits and ATM's interleaving of packet contents, there is too much state to be recorded on-chip at this time.

Packet Summarization for Software The receiver parses the cells for the various packets which are interleaved on the network connection, and reassembles the cells into packets. Once all the cells composing a packet have been received, a packet descriptor is prepended to the packet. The descriptor contains length, circuit number, checksum, and all other information that the driver may need to parse and process the packet.

Upon packet reassembly, a hardware-initiated DMA operation moves software-configured amounts of descriptor and packet contents to host memory. When

[b]The 1200-Mb/s figure includes the cost of having the application write the data to memory. Some memory bandwidth might be consumed to fill the CPU's cache in order to execute the application and operating system. In this scenario, if non-network bandwidth is greater than 300*8 — 2*1000 = 400 Mb/s, data production would be the bottleneck and the network would run at less than line rate. This is beneficial; the bottleneck has been moved to the application.

[c]Some adapters segment (or reassemble) from host memory, leading to 48-byte payload transactions with host memory. Transaction size should be an integral multiple of the cache block size, and should be aligned, in order to avoid wasting system bandwidth.

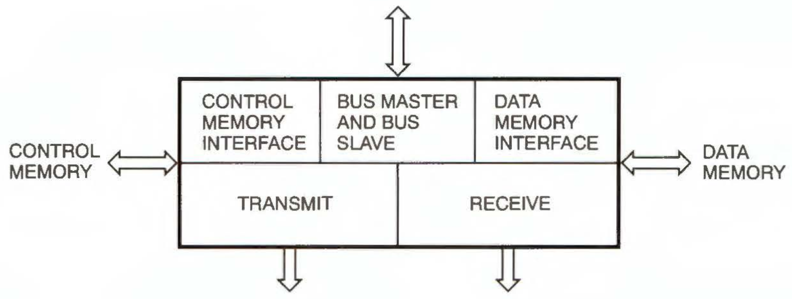


Figure 4
DART Block Diagram

properly configured, the hardware provides the network and transport headers, allowing software to determine where to place the packet data. Software data copies are avoided by allowing software to initiate a DMA operation to move the data to its final application-desired location, rather than to some expedient, but inefficient, operating system buffer.

Receive Buffering DART's store-and-forward receive buffers are divided into two classes. The per-circuit class guarantees each circuit forward progress. Each circuit is individually allocated some buffers in which to store cells. No other circuit can prevent data from passing through such buffers. The shared class is preferentially used, and avoids resource fragmentation problems. Any circuit can consume a shared buffer for an incoming cell.

Since software specifies when and where to store packet data, adapter buffers are recycled when software decides to do so, and not independently by hardware. Part of a packet may be stored in application buffers at one time, and other parts of the same packet may be stored in application buffers at later times. Hardware cannot assume a one-to-one correspondence between receive DMA and complete packet consumption.

Flow control occurs in the socket layer based on transmit buffer availability, in the transport layer based on remote receive buffer availability, in the driver based on adapter resource availability, and in the ATM layer based on cell buffer availability within the network. Credit-based flow-control protocols for ATM are based on the source of a cell stream on a link decreasing a counter (quota) when a cell is sent, and increasing a counter when a credit is received.[7] The decrement represents buffer consumption at the next hop. The credit advertises buffer availability to the source; the next hop has forwarded a cell and thus freed a buffer.[d]

[d]Forwarding the cell is required for (per-circuit) buffers of which the transmitter on the link was made aware during link initialization. The receiver on the link can generate credits immediately for (shared) buffers hidden from the transmitter during link initialization.

With FLOWmaster, the credit is conveyed across the link to the source of the cell stream by overlaying the virtual path identifier (VPI) field with the circuit to credit. This is a nonstandard optional use of the ATM cell header. Quantum Flow Control is a credit-based flow-control protocol for ATM that batches the credits into cells instead of overlaying the VPI field.

Since credit-based flow-control is based on buffer availability, credits advertising free buffers can potentially be held up by software actions. The shared class allows immediate credit advertisement, and best enables line rate communication. The per-circuit class involves software packet processing in the credit advertisement latency. To advertise a credit for a circuit whose per-circuit quota is exhausted, either the circuit must recycle an adapter-buffered packet, or any circuit must recycle a shared-class, adapter-buffered packet.

A minimal memory that constantly ran out of per-circuit buffers and flow-controlled the source would exhibit poor performance. DART uses a large data memory. Advertising (shared) buffers via credits keeps the data flowing through the overall network and systems with high performance.

Transmit Buffering Software performs all transmit buffer management. Software creates a free buffer list of its own design, allocates buffers from the list to hold packet data, and recycles buffers after observing packet completion events. Software makes the trade-off between large efficient buffers which may be incompletely filled, and small buffers which waste less storage but incur increased allocation, free, DMA specification, and transmit description overheads.

Peer-to-Peer I/O

DART avoids system resource consumption in server designs by supporting peer-to-peer I/O. A traditional server would consume PCI bus and main memory bandwidth twice by using main memory as the store-and-forward resource between two I/O devices, as shown in Figure 5. The PCI bus is consumed during steps 2 and 5. The main memory is consumed during

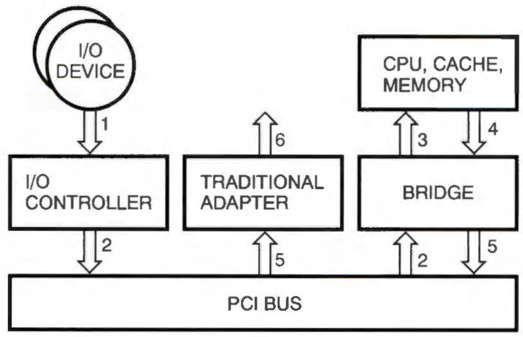


Figure 5
Traditional Server Architecture

steps 3 and 4. On some systems, I/O operations compete for cache cycles during steps 3 and 4, whether the cache is external to or internal to the CPU. Such resource consumption can cause the CPU to stall even though the CPU will never examine such data.

DART allows a single PCI bus transaction to move the data, as shown in Figure 6. This also avoids any main memory bandwidth consumption when a bridge isolates the PCI I/O bus from the main system bus. The cache is not consumed with nuisance coherence loads for data the CPU will never examine, and the CPU does not have to contend with I/O for cache or main memory cycles.

For peer-to-peer I/O over DART, the CPU is only involved in initiating packet transmission. This is a relatively small burden, since only a little bit of control information needs to be computed and communicated to the adapter.

To enable efficient peer-to-peer I/O, DART includes a bus slave as well as a bus master. *The bus slave makes the internal resources of the adapter visible on the PCI bus* through DART's PCI configuration space base address registers. Therefore, on the PCI bus, the data memory looks like a linear contiguous region of memory, just as main memory does. The bus slave supports both read and write operations for these typically internal resources.

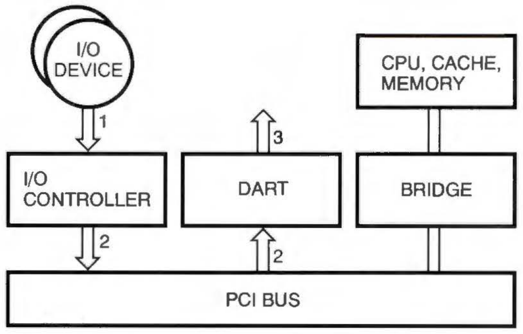


Figure 6
DART Server Architecture

DART provides efficient handling of small packets. Typically, describing a number of small packets for transmission is onerous for software, limiting the peak packet rate. DART's transmitter can automatically subdivide a large amount of data into small packets, eliminating a lot of per-packet overhead. This feature is appropriate for a video server, whose software cannot possibly fill the network pipe if it must operate on 8-cell packets.

PCI Interface

DART supports both 64- and 32-bit variants of the PCI bus. The network interface and DART memories provide prodigious bandwidth. To fully take advantage of them, a 64-bit PCI bus is recommended, but DART will also operate on a 32-bit PCI bus.

Bus Reads and Writes The DART architecture supports memory write-and-invalidate hints to the bridge between the system bus and the PCI I/O bus. Such a hint informs the bridge that the I/O device is only writing complete cache blocks. There is no need for read-modify-write operations on main memory cache blocks in such circumstances.

Write operations within a system are generally buffered. A path from the origin of the write to the final destination can be viewed as a sequence of segments. As data flows through each segment, each recipient accepts data with the promise of completing the operation, allowing each source to free resources and proceed to new operations. Thus, write paths are generally not performance-limiting as long as there is sufficient buffering to accept burst operations. In the DART context, the bridge between the system bus and the PCI I/O bus accepts DART's writes and provides buffering for high throughput.

However, read operations are more problematic. When memory locations are shared between CPUs, caches may or may not be kept coherent by hardware. Here, the memory locations are shared between the CPU and I/O device, and there is no coherence support. Each DART read suffers a round-trip time through the bridge to access the main memory. DART addresses this latency through large read transactions (up to 512 bytes).

As an example, consider a simplified 64-bit bus where 540 Mb/s of data are written in 64-byte bursts, reads suffer 15 stall cycles until the data starts to stream, and writes require a stall cycle for the target to recognize its address. Address and data are time-multiplexed at 33 MHz. Then writes consume $540 * (1 + 1 + 8)/ 8 = 675$ Mb/s of bus bandwidth. Reads have $33 * 8 * 8 — 675 = 1437$ Mb/s of bus bandwidth into which they must fit. Thus, the minimum burst length L required is $540 * (1 + 15 + L) = L \leq 1437$. The burst must be at least 9 cycles, 72 bytes, in the ideal case. DART's large read burst size compensates for overheads like large read latencies.

Importance of Bus Slave Interface The bus master interface is appropriate for software-generated transmissions. A packet created by an application in main memory can be moved via DMA to the network.

The bus slave interface is appropriate for hardware-generated transmissions. Another I/O device which is designed to always be bus master, like a disk interface, can move data directly to the DART without intermediate staging in a memory. Peer-to-peer I/O, however, was a by-product of other concerns.

Data transfer within TCP is based on a stream of large data packets flowing in one direction, and a stream of small acknowledgments flowing in the opposite direction. Traffic analysis studies often find a mix of smaller and larger packets. One of the early concerns for the DART project was to make transport protocol generation of acknowledgments inexpensive by avoiding DMA. A small packet, constructed entirely by the CPU anyway, could be moved to the I/O device instead of to main memory. This is fundamentally a short sequence of write operations that could easily be buffered, allowing the CPU to proceed in parallel on other work.

DMA from an application buffer to a device interface is generally specified to hardware by stating the physical addresses of the application buffer in main memory. DMA requires a guarantee that the data is at the specified locations. If the virtual memory system were to migrate the data to disk and recycle the physical memory for some other use, the parallel DMA activity would move the wrong data. Therefore, DMA operations are surrounded by page lock and unlock calls to the virtual memory system, to inform it that certain memory locations should not be migrated.

Additional concerns that led to incorporation of the bus slave interface were related to the cost of page locking, and the cost of acquiring and releasing DMA resources (e.g., in the bridge). An acknowledgment might be constructed in nonpaged kernel memory, but a small application packet would likely be constructed in application memory subject to paging. Even if page locks were cached for temporal locality, it might be cheaper to simply move the data via programmed I/O.

The break-even point between DMA and programmed I/O is system-dependent, but can be measured at boot time in order to learn an appropriate threshold to use for such a decision. Demands on the main memory system from its various clients will change over time, and a single measurement is only optimal for the sample's conditions. The suggestion here is to enable a quick judgment in the software. The intent is to make large gains and avoid egregious performance errors. We suspect that fine-tuning the decision is less important, and requires the collection of excessive information during the normal operation of the system.[c]

Interrupt Strategy As noted above, on-chip access rates for the CPU increase more quickly than off-chip access rates. Interrupt processing and context switching are fundamentally off-chip actions; new register values must be loaded into the CPU, and the cache must be primed with data. Thus, the general system trend is that interrupt processing and context switching improve more slowly than raw processing performance.

DART provides a programmable interrupt holdoff mechanism. By delaying interrupts, events can be batched to reduce various system overheads. If the batching mechanism were not present, an interrupt per packet would swamp system software at gigabit rates.

Since the interrupt delay interval is programmable, software may use adaptive algorithms to decrease interrupt latency if the system is idle, or to increase the amount of batching if the system is busy. The delay timer starts decrementing as soon as it is written. Typically, the timer will be written at the end of the interrupt service routine.

Interrupts can be divided into two classes by software. Each class has its own delay interval, in case software assigns distinct importance or latency requirements to the classes.

The Dart Software

DART provides increased performance with the same system calls, and with the existing system call semantics. The only change is to the underlying implementation of the existing system call semantics.

Unmodified existing applications can consume gigabit network bandwidth. The application can assist the system software by using large contiguous data buffers, but it is not required. System software can specify byte-level scatter/gather operations to the DART adapter in order to access arbitrary application buffers.

Changes to the system software are confined to a few locations above the driver layer, and are generic. Successive high-bandwidth adapters for other media can be supported by just writing drivers; no changes will be needed above the driver layer. The shared set of upper-layer software changes are only needed to take maximum advantage of a DART-style adapter; a traditional copy-based implementation is supported by the hardware.

[c]Given the parallel nature of the environment (other I/O, cache operations, and multiprocessor CPUs), a software system could only estimate non-DART memory loads. Queued DMA operations may start later than expected, or finish before their completion has been noticed. CPU cache activity is dependent on the program executing at that moment; fine-tuning is problematic. The focus of DART has been the large gains, like avoiding copies, or allowing either DMA or programmed I/O to be used. The focus has been on the structure of the system.

We developed a prototype UNIX driver to test the upper-layer changes, and executed a modified kernel against a user-level behavioral model of a DART-style adapter. The code was subjected to constant background testing on a workstation relied on for daily use. The prototype driver supports buffer descriptors referencing either kernel buffers or adapter buffers. The implementation effort to support kernel-buffered packets was minimal, and enables multiple protocol families to be layered above the driver.

The software changes modify the existing upper-level software, rather than bypassing it via a collapsed socket, transport, network, and driver implementation. The current UNIX networking subsystem provides a rich set of features that needs to be completely supported for backward compatibility.

Transmit Overview

A comparison of traditional transmission with DART transmission is shown in Table 1. For a traditional adapter, the system call layer copies application data to operating system buffers. With a DART adapter, the data is copied to the adapter. Uiomove is the copy function typically used within UNIX. The DART mechanism is to use an indirect function call through a pointer, rather than a direct function call to an address specified by the compiler's linker. High-performance copy functions are associated with the device driver. The driver's copy function is free to use DMA or programmed I/O, depending on the length of the copy.

For a traditional adapter, software wastes machine resources computing checksums. With a DART adapter, the checksum is computed by hardware as the data flows into the adapter. The adapter can patch the checksum into the packet header. The adapter can also move checksum summaries back to host memory so that they are available for retransmission algorithms.

For a traditional adapter, the driver instigates additional memory references to copy the data to the adapter for transmission. With a DART adapter, the data is already on the adapter, ready to be sent! Much of the data copy avoidance work is throughput-related. In this instance, we also create the potential for a latency advantage for the DART model, since the data copy overlapped work in the system call, transport, network, and driver layers of the operating system.

Receive Overview

In many ways, the receive path for networking is usually considered more complicated than the transmit path, since the various demultiplexing and lookup steps are based on fields that historically have been considered too large to use simple table indexing operations. Also, the receive path requires a rendezvous between the transport protocol and the application (to unblock the application process upon data arrival). So it should come as a pleasant surprise that the DART-style changes for packet reception can be as simple and localized as two conditionals in the socket layer and one in the network transport layer.

Table 2 is a comparison of traditional receive processing with DART receive processing. It is almost identical to the packet transmission comparison. The distinction is which portion of the DART adapter computes the checksum on behalf of the software (receiver instead of DMA engine).

Interrupts

Transmit completion interrupts do not need to be eagerly processed. Software can piggyback processing to reclaim transmit buffers upon depletion of transmit buffer resources, upon unrelated packet reception events (e.g., User Datagram Protocol, UDP), and upon related packet reception events (e.g., TCP acknowledgment). The transmit completion events can be masked, or the hardware interrupt holdoff mechanism can be used to give them a longer latency.

Receive interrupts are batched to reduce overheads. Short packets are fully contained in the initial packet summary which would be deposited in a kernel buffer. Adapter buffers for short packets can be recycled immediately by system software. Long packets are not fully contained in the initial packet summary provided software for parsing and dispatch. The summary is noticed during one interrupt, and scatter/gather I/O completion into application buffers is noticed during another interrupt if performed asynchronously.

The side-effect of the decision to create a store-and-forward adapter is that a received packet is related to two interrupts. The intent is *not* to burden a system and cause multiple interrupts per packet. The distinction between *relation* and *causality* is important.

When the system is under load, there is a steady stream of packets, and thus a steady stream of batched

Table 1
Transmit Overview

	Traditional	DART
System call layer	Uiomove user buffer to kernel buffer	*Uiomove user buffer to adapter buffer
Protocol layer	For all buffers for all bytes, update checksum	For all buffers, update checksum
Driver layer	Programmed I/O or DMA	Data is already on the adapter!

Table 2
Receive Overview

	Traditional	DART
Driver layer	Programmed I/O or DMA	Data stays on adapter!
Protocol layer	For all buffers for all bytes, update checksum	Use checksum computed by receiver hardware as packet was reassembled
System call layer	Uiomove kernel buffer to user buffer	Uiomove adapter buffer to user buffer

interrupts. If 3 Mbytes were transferred using a burst of 1-kbyte packets, there would be 3000 packets. Batching 20 packets/interrupt, there would be 150 interrupts to report packet arrivals. The first interrupt is just for packet arrival events, to allow header parsing. The intent is for the next 149 interrupts to report 20 new arrivals and the DMA completion for 20 previous arrivals. A final interrupt would take care of the final DMA requests. In this case, the additional interrupt load for a DART adapter is minor: one interrupt for 3000 packets. The interrupt load is not doubled (even if one chooses to move received data asynchronously).

Store-and-forward latency is incurred because of the memory write and read on the adapter (to store data from the network and to later move it to the application's buffers). DART adapter memory operates at a high rate, over 4 Gb/s, to minimize this. Due to the intervening software decision concerning where to place DART data for large packets, the data may be placed at its initial location in host memory later than for a traditional adapter which fills kernel buffers. However, store-and-forward reduces main memory bandwidth consumption, and quickly places the data at its *final* location within the application buffers in host memory. The correct metric is latency to data availability to the application, not data latency to first reaching the system bus.

CSR Operations

Control and status registers (CSRs) are used within hardware implementations to allow software to control the action of hardware, and for hardware to present information to software. For example, a CSR can inform a device of the device's address on a bus. In this case, the CSR's definition is generic in the context of the bus definition. Alternatively, a CSR can be used to initialize a state machine within the hardware implementation. In that case, the CSR's definition is specific to that version of the device.

CSR reads are very expensive. Generally, a single CSR read is required for DART interrupt processing, and that CSR is placed in the PCI clock domain of DART in order to avoid operation retries on the PCI bus.

Most packet processing information is written to host memory by the adapter for quick and easy CPU access. For example, packet summaries are placed in one or more arrays in host memory, and software can use an ownership bit in each array element to terminate processing of such an array.

CSR writes are buffered; nevertheless, they can be minimized. The packet summaries in host memory are managed with a single-producer, single-consumer model. When the consumer and producer indices into an array are equal, the array is empty. When hardware's producer index is greater, there are entries to be processed by software. (Redundant information in array element ownership bits means that software does not actually need to read the DART adapter to perform the producer-consumer comparison.) When the hardware's producer index reaches the software's consumer index minus one, the array is fully utilized. When software has processed a number of packet summaries, the hardware can be informed that they can be recycled by a single write of the consumer index to the adapter.

The DMA engine processes a list of "copy this from here to there" commands. By supporting a list of operations instead of a single operation, software can quickly queue an operation and move along to its next action without a lot of overhead. The copy commands reside in an array within host memory, with a software-specified base and a software-specified length.

DMA commands also follow the producer-consumer model. However, since instructions are only read by DART, there are no ownership-bit optimizations. To compensate for this, software can allocate a large array and cache a pessimistic value for the hardware's consumer index in order to avoid CSR reads. Alternatively, the DMA engine could periodically be given instructions to DMA such information to host memory.

A typical DART interrupt involves one CSR read and three CSR writes, yielding an efficient interface. One read determines interrupt cause. One write informs the DMA engine of new copy commands for newly received data. Another write informs the DMA engine that the CPU processed a number of the packet summaries DART placed in main memory. A third write initializes the interrupt delay register to batch future events.

Occasionally, an interrupt also involves an extra CSR read. The read discovers a large number of commands processed by the DMA engine, allowing software to recycle entries in the command queue and thereby issue more commands.

Driver

The driver classifies received packets, and decides whether to continue to use adapter buffers for them, or to copy the data into kernel buffers. For the prototype, adapter-buffered packets are:

- Long enough to contain maximal-length IP and transport protocol headers.
- Version 4 IP packets (buffering assumptions percolate throughout the layers of the system, so a protocol family must be updated and tested to support adapter-buffered packets).
- TCP or UDP protocol packets. Other protocols layered over IP do not use adapter buffers, to make the scope of the effort manageable by handling just the common case.

The operating system uses a single *mbuf* to describe a single set of contiguous bytes in a buffer which may be within or external to the mbuf structure. Mbufs can be placed in lists to form packets from a number of noncontiguous buffers.

Received adapter-buffered packets are two mbufs long. The first mbuf contains the initial contents of the packet DMAed into memory by the adapter, that is the protocol headers and summary information from the adapter.

The second mbuf refers to the packet in adapter memory. For ATM, the received packet is stored in a linked list of buffers on the adapter. Programmed I/O access to the buffers requires software to traverse the links, but this would not be done in practice since the CPU read path to the I/O device is unbuffered and high-latency. The DART DMA hardware would be used, and it would traverse the links as-needed. The DMA hardware allows the software to pretend the packet is contiguous.

Fields of the second mbuf are used in specific ways. The length of the second mbuf does not contain the initial portion of the packet copied into the first mbuf, even though the adapter memory buffers the entire packet. The initial portion is replicated, but only the copy local to the CPU is accessed. The pointers of the second mbuf point to bogus virtual addresses, even though the adapter looks like an extension of main memory. This speeds software debugging by trapping inefficient accesses to the adapter. Adjusting the length and pointer fields is still allowed in order to drop data from the front or back of the mbuf. The m_ext fields record the location and amount of adapter buffering used to hold the packet. They also point to a driver-specific buffer reclamation routine.

For TCP, or for UDP packets with nonzero checksums, the driver makes incremental modifications to the DART receive hardware's checksum. The hardware computes the 1's complement checksum over all the cell payloads except for the final ATM trailer bytes. As a result, the driver modifies the hardware checksum to account for:

- Contributions made by IP options
- Construction of the pseudo-header which is not transmitted on the network
- The transport layer checksum, which was zero when the checksum was computed but may be nonzero on the network

To transmit a packet, the transport and network layers operate on protocol headers in main memory. The driver moves the headers to the adapter as part of transmitting a packet whose encapsulated data is in adapter buffers.

The *ifnet* structure is the interface between the protocol layers and the driver. It contains, for example, fields expressing the maximum packet size on the directly connected network, the network-layer address of the interface, and function pointers used to enter the driver.

We add an *(*if_ uiomove)()* field to be associated with buffers as described below. It represents a driver entry to copy data to or from the adapter. We also add an *(*if_ xmtbufalloc)()* field to be used within the mbuf allocation loop of the transmit portion of the socket layer. This allows the socket layer to give precedence to allocating (large) adapter buffers over main memory buffers.

The driver always retains some transmit adapter buffers for its own use. When the system is busy, there will be TCP packets consuming adapter buffers. The packets are associated with the socket send queue. There will also be packets on the interface send queue, which may or may not use adapter buffers. If the first item on the interface queue uses just kernel buffers, then the driver must have reserved adapter buffers in order to complete the transmission and avoid transmit deadlock. At least one packet of adapter buffering must be reserved for the driver output routine.

UDP

UDP motivates many of the changes without getting involved in the complexity of retransmission and reliability. Many of these changes are generic to UDP and TCP: augmenting the buffer and interface descriptions, discovering the availability of efficient buffers for a connection, and allocating and filling the efficient buffers.

One portion of the mbuf is the *struct pkthdr,* which is used only in the first mbuf of a packet. It summarizes interesting information about the packet, like its total length.

We add a *protocolSum* field to the pkthdr of the mbuf so that the driver can communicate the received transport-layer checksum to the upper layers. The transport-layer checksum is not ignored, as it would be if checksums were negotiated away or cavalierly disregarded. The checksum is verified by the transport layer as usual, but without accessing all the bytes of the packet. The protocolSum field is valid if an *M_PROTOCOL_SUM* bit is set in the mbuf m_flags field.

Another portion of the mbuf is the *struct m_ext,* which is used to describe data buffers external to the mbuf structure. We add an *(*uiomove_f)()* field so that the driver can communicate a buffer- or driver-specific copy routine to the socket layer. Socket layer usage of the standard pre-existing uiomove routine assumes that the received data is in the address space and should be moved by CPU byte-copying. The indirection allows the data to be moved by programmed I/O or DMA. The uiomove_f field is valid if an M_UIOMOVE bit is set in the mbuf m_flags field. Parameters to the uiomove_f function are an mbuf, an offset into the packet at which to start copying bytes, a number of bytes to copy, and the standard uio structure that describes where the application wants the data.

The UDP input routine performs protocol processing on received UDP packets. Before the pseudo-header is constructed for checksum verification, the M_PROTOCOL_SUM bit is tested in order to skip CPU-based checksumming.

```
if (m->m_flags & M_PROTOCOL_SUM) {
  NETIO_COUNT(rch_hw_sum);
  assert(m->m_flags & M_PKTHDR);
  if (ui->ui_sum != m->m_pkthdr.protocolSum) {
    NETIO_COUNT(rch_hw_sum_bad);
    goto badsum;
  }
  goto ok;
}
```

Error processing can be based on packets reformatted into kernel buffers. The UDP output routine performs protocol processing on transmitted UDP packets.

Checksum overhead avoidance is similar to the receive path; but instead of testing the M_PROTOCOL_SUM bit, the mbuf checksum field is assumed to be valid for all transmit mbufs referencing adapter buffers (they have the M_UIOMOVE bit set). We assume that no adapter which saves the operating system the effort of data copying would forget to save the operating system the effort of checksumming. It does not make sense to eliminate some, but not all, of the per-byte overhead operations.

For UDP transmission, software recycles (adapter) buffering after the packet has been transmitted.

Changes like checksum avoidance are based on adding a conditional to the existing code paths. For a DART adapter, the test and branch penalty are small relative to the gain. For large external buffers, there are one or two M_PROTOCOL_ SUM tests per packet, depending on packet length and buffer size. This could be viewed as a constant-time overhead. The gain is avoiding the linear-time access of each byte within each packet.

For a traditional adapter, the test and branch represent overhead for each packet. The cost of the added conditionals occurs in the context of a large code base between the system call interface and the driver, and that networking code provides a rich feature set through the use of conditionals. If the added conditionals are viewed as significant, consider the approach of generating two binary files from a single source module. To avoid penalizing systems populated solely with traditional adapters, operating system software configuration procedures can choose not to incorporate the DART-conditionalized version of the code. A DART adapter installed at a later date would still operate under such a software configuration, but would not reach its peak performance until the software is reconfigured to use the DART-conditionalized version.

TCP

The TCP input routine performs protocol processing on received TCP packets. Before the pseudo-header is constructed for checksum verification, the M_PROTOCOL_SUM bit is tested in order to skip CPU-based checksumming. The only differences with the UDP input processing change are the names of the TCP header structure and TCP header checksum field.

All the adapter resources represented by the second mbuf of a received packet are consumed until the final reference to the packet is freed. If large packets are exchanged and the application is doing small reads, not until the final read is any storage reclaimed. This space consumption is represented on the socket receive queue, and therefore affects the advertised TCP window.

The TCP output routine performs protocol processing on transmitted TCP packets. The checksum overhead avoidance is similar to that done for UDP. Checksum computations for transport-layer retransmissions are simplified by the association of checksum contributions with mbufs, rather than an association of checksums with packets. The association with buffers instead of packets also simplifies handling of packets using a mix of kernel and adapter buffers.

For TCP transmission, software recycles (adapter) buffering after the packet has been acknowledged by the remote end of the connection. Between transmission and acknowledgment, the data is held on the socket's send queue. Previously, the socket code copied data from one mbuf into another whenever both mbufs' contents fit into one, trading increased CPU load for space efficiency. For DART adapters, the copy decision is cut short.

We add a *bytesSummed* field to the mbuf so that when a packet is transmitted or retransmitted by the transport layer, code can double-check that all the data the checksum is supposed to cover is still present in the buffer. For example, a TCP acknowledgment of part of an original packet generally leads to the sender deleting its copy of the acknowledged data retransmitting the rest. The software implementation handles the generality of acknowledgments which are not complete transmit mbufs, the unit covered by the protocolSum field. A retransmission must not send a packet with an improper transport-layer checksum, even if it means using an algorithm linear in the number of bytes remaining in the buffer to recompute the checksum.

The transmitter's socket layer buffers data in segments convenient for both the network-layer protocol and the driver. Checksum contributions remembered for retransmission are recorded at a similar level of granularity. The transmitter is liberal in what the receiver can acknowledge; the receiver's implementation affects efficiency, but not correctness.

Socket Data Movement

The copy from the network buffers to the application data space occurs in the *soreceive* routine, which uses information left in the mbuf by the device driver. The call(s) to uiomove become conditionalized as follows:

```
if (m->m_flags & M_UIOMOVE) {
  assert(m->m_flags & M_EXT);
  error = (*m->m_ext.uiomove_f)(m, moff, len, uio);
} else
  error = uiomove(mtod(m, caddr_t) + moff, len, uio);
```

The reverse copy in *sosend* is similar.

The standard uiomove function makes the optimistic assumption that the addresses of user buffers provided by the application are valid. If addresses are not valid, a trap occurs and situation-specific code is called.

To support drivers that use programmed I/O movements with the application's buffer, an additional code point is added to the error processing so that an EFAULT error is returned to the application.

Note that the changes are generic, and can be used with existing devices. The uiomove_f function can perform both copies to kernel buffers and protocol checksumming for transmission over traditional adapters.

In the transmit portion of the socket layer, the application data is moved to kernel buffers or to adapter buffers by sosend. In order to take advantage of DART adapters, sosend needs to know:

- That the protocol layers between the socket and driver support DART-style buffering
- That the driver supports DART-style buffering

In general, formatting data efficiently for transmission can require knowing the amount of headers that will be prepended by the various layers below the socket layer, so device alignment restrictions can be met. Due to protocol options and to the variety of media in existence, the amount prepended may vary from socket to socket. Given a socket, we introduced a function that computes:

- A function pointer for allocating adapter-based buffers
- A function pointer for moving data from user buffers to adapter buffers
- The number of bytes required to prepend all headers

To simplify the prototype implementation effort, the function disallows the use of adapter buffers for IP multicast packets.

When allocating adapter buffers, sosend uses the *if_xmtbufalloc* entry to allocate adapter buffers. Each time it does so, it passes a maximum number of bytes of buffering that attempts to allocate a buffer for the entire (remaining portion of the) packet. The driver indicates the actual amount of buffering allocated; sosend loops until all the necessary buffering is allocated. The driver may decline to allocate an adapter buffer if the requested amount of buffering is small. At that time the driver can best decide if CPU-based byte copying from user buffers to kernel buffers, and also copying kernel buffers to the adapter, is preferable to programmed I/O or DMA from user buffers.

Once an adapter buffer allocation fails, no further allocations are attempted within a segment that will be passed to the lower layers. This ensures that drivers will see, at worst, an (internal) mbuf containing headers, one or many adapter buffers containing data, and potentially one or many kernel buffers containing the rest of the packet. This simplifies the driver, and ensures that alignment restrictions are met without shuffling data around on the adapter. It also simplifies transport-layer checksum computation algorithms.

There is an unusual boundary case in which a long segment of transmit data may not immediately be copied to adapter buffers, even though the driver would prefer to do so. If the driver has many free transmit adapter buffers when the socket code starts to prepare a segment, it may not have any free buffers when the segment nears completion. This is because the socket layer runs at a lower interrupt priority level than the device driver, and buffers are allocated individually. A device interrupt can lead to servicing the device output queue, consuming adapter buffers in order to transmit traditional kernel-buffered packets. Rather than block and wait for transmit adapter buffer availability, the prototype software uses kernel buffers.

Both the socket and network protocol (TCP) layers contain segmentation algorithms. In the socket layer, the segmentation process is confused with the (cluster mbuf) buffer choice decision procedure. As part of eliminating that confusion, we introduce an *if_buflen* field to the ifnet structure.

If the socket layer creates segments longer than the device frame size, excess work occurs in the lower layers (e.g., TCP segmentation or IP fragmentation). If the socket layer creates segments shorter than the device frame size, the system foregoes large packet efficiencies. A large 8-kbyte write that leads to eight 1-kbyte cluster mbufs being individually processed by the lower layers might benefit from overlapped I/O of the first segment with computation of the last, but the CPU would be wasted for a benefit that is only relevant when a large number of such poorly chosen segments are constructed. Such a write could go out as a single packet over an ATM network.

Socket Buffering and Flow Control

A number of papers have commented on the requirement for a reasonable amount of socket buffering to enable applications to "fill the pipe" with a "bandwidth times delay" amount of data.[1] Delay includes the link distance, device interrupt latency, software processing, and I/O queuing delays. It also includes interrupt delays that aggregate events for efficient software processing.

The requirement for sufficient socket buffering is a lesson learned over and over again. Traditional solutions include marginal increases in systemwide defaults, and application modification to request more buffering than the default. Facilities like rsh imply that anything can become a network application, unbeknownst to the application author; so changes to applications are a poor solution. Also, applications are insulated from the network by the network protocol and socket abstractions; no application should need to know the buffering requirements for high throughput for the media *du jour.*

We introduce an *(*if sockbuf)()* entry that allows the driver to increase socket buffering. When local network-layer addresses are bound to socket connections, an interface is associated with the connection, and the driver is allowed to adjust the socket buffer quota.

For TCP server connections, the server may not be restricting incoming connections to a particular interface. Overriding the default buffering value must be done on the socket created when the incoming SYN arrives, not on the placeholder server socket. The buffer allocation needs to be determined as soon as possible, because the initial SYN packet also triggers the determination of the proper window scaling value.

UDP does not queue packets on the socket send queue. Although calls to *if_sockbuf* from the socket layer are independent of the protocol, the buffer quota only affects the maximum UDP packet size sent, not the number of UDP packets that can be in flight at the same time. The socket is not charged for UDP packets queued on the driver output queue or UDP packets in the hardware transmit queues.

The adapter buffer resources are distinct from main memory mbuf and cluster resources. The socket data structure and support routines support consumption and quota numbers for adapter buffering that are distinct from the current main memory consumption and quota numbers. For example, a connection redirected from a DART adapter to a traditional adapter is quickly flow-controlled in the socket layer as a result. The large adapter buffer allocation does not enable it to hog main memory buffers and adversely affect other connections.

IP

The prototype software contains conditionals to enable or disable the use of adapter buffers for messages undergoing IP fragmentation. This only affects UDP, since the socket layer segments appropriately for the TCP and driver layers. Software computes the amount of header space for the first fragment, and also the amount of header space for the following fragments (which will not contain transport protocol headers). This information is used during the socket layer's movement of application data to kernel or adapter buffers. UDP and IP receive the segments as a single message; the IP fragmentation code uses the fragment boundaries precomputed in the socket layer.

IP reassembly of received adapter-buffered packets was implemented in the prototype code to keep up with a transmitter using adapter buffers for IP fragmentation. The driver adjusts the hardware-computed checksum to ignore the contribution to the hardware sum caused by the successive IP fragment headers, which are not presented to the transport layer.

Resource Exhaustion

The hardware provides a scalable data memory. The memory holds received data until the application accepts it, and transmits data until the acknowledgment arrives. The prototype provides 16 Mbytes, which was considered a significant quantity after examining network subsystem buffering at centralized servers for several large "campus" sites.

When adapter memory is scarce, it should be allocated to connections whose current data flows are high-bandwidth flows. Low-bandwidth connections, connections blocked by a closed remote window, and connections over extremely loss-prone paths will not be significantly impacted by the copying overhead associated with the use of kernel buffers.

Data Relocation

Reformatting data from adapter buffers to kernel buffers allows existing code to be ignorant of adapter-buffered data. Socket-based TCP communication can use adapter buffers for high throughput, and other protocol environments can simultaneously use the familiar kernel buffers. DART support can be phased in by protecting legacy code with a conditional relocation call before entering or queuing data to the legacy code. Cache fill operations should be targeted to main memory, not adapter memory, for best performance in legacy code.

Relocation is also appropriate for error handling and other rarely executed code paths. For example, a multi-homed host may lose TCP connectivity through the first-hop router associated with a DART link, and be forced to send packets over another link. The new communication path could use any network interface, DART or otherwise. The software needs to be able to handle the scenario where the new adapter, or some system resource, has a constraint preventing it from transmitting packets located in DART memory.

We selected a lazy evaluation solution which assumes that data sent over an old route will be delivered and acknowledged. An eager solution would incur a large burst of data relocation when the new route takes precedence, with the disadvantages that the work would be wasted for data which is acknowledged, and the burst of activity consumes resources and incurs increased latency for other activities.

For TCP connections marked as using adapter buffers, a driver entry through *(*if_ pktok)()* allows the driver to comment on each outgoing packet. This implies that the driver also comments on TCP retransmission packets. The driver has a chance to double-check constraints and trigger data relocation, if necessary. Drivers not supporting *if_pktok* always trigger data relocation, and also lead to unmarking the TCP connection.

Comparison to Other Methods

Traditional adapters contain minimal onboard memory and hide their buffering from the CPU. Unable to manage a traditional adapter's buffers, a copy of data must be kept in host memory until it is acknowledged in case it needs to be retransmitted.

We felt copy-on-write approaches to using a traditional adapter would be inadequate due to bookkeeping overheads experienced by other projects. Also, the application may commonly reuse the same application buffer before the transport protocol semantics allow. For an unmodified application, this would lead to blocking the application, or incurring both copy-on-write and data copy overheads. All applications are network-based when one considers networked file systems and pipes to remote program invocations; architectures that require applications to be recoded to interact with page mapping schemes (e.g., [8]) are inadequate. Another objection is that copy-on-write focuses on packet transmission, ignoring packet reception.

When a write is performed by an application using DART, the application blocks only long enough to buffer the data, as for a traditional adapter. The copy of the application's data on DART enables retransmission for reliable communication. The application is free to immediately dirty its write buffer, and no performance impact is associated with that action.

Van Jacobson's WITLES paper design uses the CPU to copy data to and from the adapter via programmed I/O.[9] Reading the adapter is an expensive operation, and in practice would provide worse receive performance than even a traditional adapter. The Medusa design is a WITLES variant that uses programmed I/O transmission and addresses the receive penalty with system block-move resources for reception.[10] The Afterburner design used the same approach, achieving 200 Mb/s.[4] The WITLES approach keeps the packet in adapter memory until it is copied to the application buffer.

To minimize resource consumption, the checksum and copy loop are combined. This means that the TCP acknowledgment is deferred until the application consumes the data, which might be much later than necessary. Applications read data at a rate of their own choosing. Care must be taken that this deferral does not lead to TCP messages to the data source that cause unnecessary data retransmission.

Unlike WITLES, DART supports DMA to and from the adapter. Software can use DMA where appropriate, intelligently balancing the costs of programmed I/O and DMA.

Since DART provides the IP checksum with the packet, the TCP acknowledgment can be sent as soon as the packet is reassembled and reported to the CPU. The acknowledgment contents and transmission time are traditional BSD UNIX; it states that the data has been received, and the offered window reflects buffer consumption until the application receives the data at its leisure.

Adapters have been built that offload protocol processing.[1] However, the cost of TCP processing is low, and such an architecture introduces message-passing overheads that counterbalance the offloaded protocol processing efficiencies. CPU execution rates are scaling well. The issue to address is the main memory bandwidth bottleneck. Also, it is expensive and difficult to create, maintain, and augment the firmware for such an adapter. The firmware is tied to a single adapter, and replicates work done within the operating system that can be shared by a number of adapters.

DART provides assist via checksumming methods. It does not attempt to offload network- or protocol-layer processing.

Performance

The simulation environment used to debug and test the chip design was also used to extract performance information. The chip model used to fabricate the part is connected to a PCI bus simulation, some generic bus master devices, and some generic bus slave devices. The simulation environment is connected to and controlled by a TCL-based environment.

Within the TCL environment, the hardware designers wrote a device driver. With this driver, DART copied packets from host memory, looped packets on an external interface, reported packet summaries, and copied packets into host memory. Both 64- and 32-bit PCI buses were exercised. Target read latency of host memory was incorporated into the simulation (the data presented in Figure 7 is based on a 16-cycle latency). Credit-based flow-control operations were enabled since they consume additional control memory bandwidth, and therefore represent worst-case-scenario operation, Similarly, a large number of virtual circuits were used to loop data, to prevent the use of on-chip, cached circuit state.

Because the TCL driver was written by hardware designers, and they were focused on designing and testing the chip, performance numbers extracted from their work suffer from a lot of CSR accesses. A real driver would reduce the CSR operations and have increased batching of interrupts and other actions.

CSR reads are costly, since they involve a round-trip time within the chip which crosses clock boundaries, in addition to the round-trip time between the CPU and the pins on the device. Crossing clock boundaries means that there are internal first-in first-out (FIFO) delays involved to deal with synchronization and meta-stability issues. To meet PCI latency specifications, the bus master is told to retry such operations, freeing the PCI bus for other use during the internal round-trip time. CSR writes are efficient, since they are buffered throughout the levels of the system.

The dip in Figure 7 is near the 512-byte burst size used to read from host memory. Packet transmissions no longer fit in a single DMA burst, and incur the extra cost of an additional short fetch. This incurs additional overhead cycles to place the address on the bus and for the target to start to respond with the first bytes.

For each simulation we extract numerous detailed statistics. Table 3 contains a few for 32-cell packets (1536 bytes) on a 32-bit PCI bus. These particular figures are for the TCL driver, and include time intervals to initialize the adapter, to transmit before the first packets are received, and to receive after the last packet was transmitted.

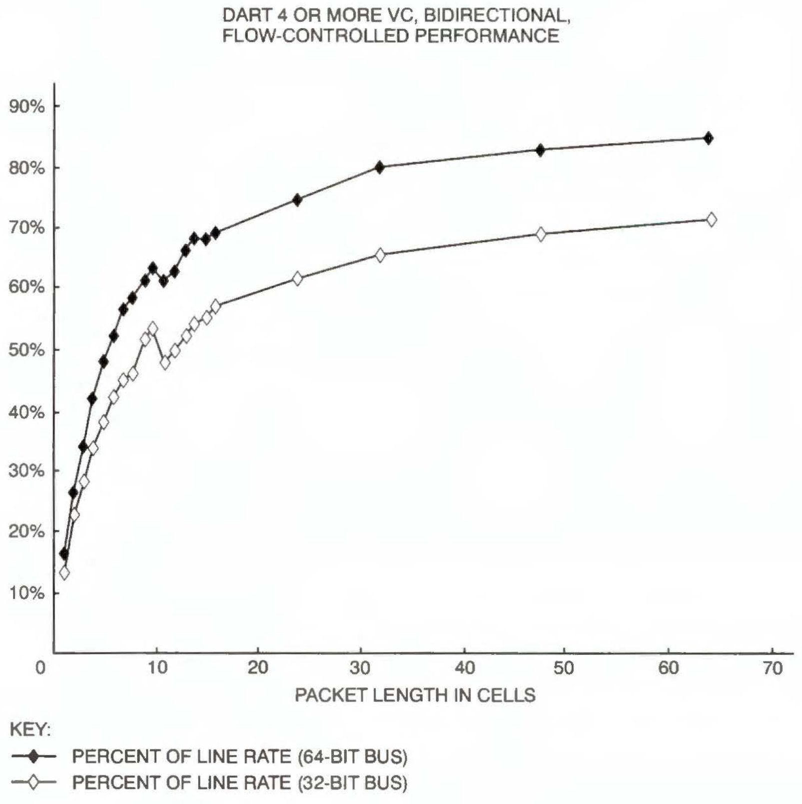


Figure 7
DART Performance

Table 3
Examples of Additional Statistics

Control memory idle	79%
Data memory idle	48%
PCI busy (frame or irdy asserted)	75%
PCI transferring data (irdy and trdy asserted)	60%
CSR operations share of bus operations	41%

Future Work

Due to the large amount of onboard buffering, we do not expect DART to encounter resource exhaustion issues. However, some work will be appropriate to determine the best solution should buffering requirements exceed the electrical capabilities of the high-speed SAR-SDRAM interface. Is it efficient to move unacknowledged data off the adapter so that new transmit data can be moved from user space to the adapter in the socket layer? Is it efficient to block in the socket layer, waiting for adapter buffers to be freed by a future, or arrived but unprocessed, acknowledgment? Is it efficient to use conventional kernel buffers to transmit when the space allocated to DART-style transmissions is exhausted?

DART structures the system software so that the operating system does not examine the application's data, which should be private to the application anyway. This separation of control operations (on headers) from data operations (primarily movement) is a common theme in embedded system design for bridges and routers. DART provides a generic structure that enables high-performance networking in a variety of systems.

With features like peer-to-peer I/O, one can conceive of a system with multiple gigabit links, where the bottlenecks have shifted from the system software to the application or service. We think DART-style adapters will enable and accomplish the delivery of high-bandwidth service to the application.

Acknowledgments

Robert Walsh implemented the transmitter and PCI bus interface. Kent Springer implemented the receiver and packet reporting functions. Steve Glaser implemented the DMA engine. Tom Hunt implemented the external control RAM interface, the external data RAM interface, and the board design. Robert Walsh developed the prototype UNIX changes. Phil Pears, Mark Mason, James Ma, and Ken-ichi Satoh provided significant assistance in placing and routing the ASIC.

We also had assistance from Joe Todesca, Elias Kazan, and Linda Strahle. Bob Thomas participated in the initial concept and design. K.K. Ramakrishnan provided some information on networking performance.

References

1. Metcalfe, "Computer/Network Interface Design: Lessons from Arpanet and Ethernet," *IEEE JSAC,* vol. 11, no. 2 (Feb. 1993).
2. Walsh and Gurwitz, "Converting the BBN TCP/IP to 4.2BSD," *USENIX 1984 Summer Conf. Proc.* (June 1984).
3. Chang et al., "High-Performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP," *Digital Technical Journal,* vol. 5, no. 1 (Winter 1993).
4. Dalton et al., "Afterburner," *IEEE Network* (July 1993).
5. Clark et al., "An Analysis of TCP Processing Overhead," *IEEE Commun. Mag.* (June 1989).
6. Kay and Pasquale, "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000," *USENIX 1993 Winter Conf. Proc.* (1993).
7. Owicki, "AN2: Local Area Network and Distributed System," *Proc. 12th Symp. Principles of Dist. Comp.* (Aug. 1993).
8. Smith and Traw, "Giving Applications Access to Gb/s Networking," *IEEE Network* (July 1983).
9. Van Jacobson, "Efficient Protocol Implementation," ACM SIGCOMM 1990 tutorial (Sept. 1990).
10. Banks and Prudence, "A High-Performance Network Architecture for a PA-RISC Workstation," *IEEE JSAC,* vol. 11, no. 2 (Feb. 1993).

Additional Reading

1. Kay and Pasquale, "The Importance of Non-Data Touching Processing Overheads in TCP/IP," *Proc. SIGCOMM '93 Symp. Commun. Architectures and Protocols* (1993).
2. Ramakrishnan, "Performance Considerations in Designing Network Interfaces," *IEEE JSAC,* vol. 11, no. 2 (Feb. 1993).

Biography

Robert J. Walsh
Robert Walsh has been working on high-speed networking since the beginning of the 1980s. He developed networking software for BSD UNIX, BBN's Butterfly multiprocessor, and DIGITAL's GIGAswitch/FDDI.

Recent DIGITAL U.S. Patents

The following patents were recently issued to Digital Equipment Corporation. Titles and names supplied by the U.S. Patent and Trademark Office are reproduced as they appear on the original published patent.

D391,927	Robert T. Faranda and Bradford G. Chapin	Notebook personal computer
5,596,218	Hamid R. Soleimani, Brian Doyle, and Ara Philipossian	Hot carrier-hard gate oxides by nitrogen implantation before gate oxidation
5,596,283	Richard I. Mellitz and Michael V. Dowd	Continuous motion electrical circuit interconnect test method and apparatus
5,596,575	Henry S. Yang, Donald L. Post, and Wen-Yi Huang	Automatic network speed adapter
5,596,715	Philippe Klein, David W. Maruska, and Kevin W. Ludlam	Method and apparatus for testing high speed busses using gray-code data
5,596,754	David B. Lomet	Method for performing private lock management
5,608,653	Ricky S. Palmer and Larry G. Palmer	Video teleconferencing for networked workstations
5,608,883	Robert R. Kando and Paul L. Godin	Adapter for interconnecting single-ended and differential SCSI buses to prevent 'busy' or 'wired-or' glitches from being passed from one bus to the other
5,614,444	Janos Farkas, Rahul Jairath, Matt Stell, and Sing-Mo Tzeng	Method of using additives with silica-based slurries to enhance selectivity in metal CMP
5,615,167	Anil K. Jain, John H. Edmondson, and Peter J. Bannon	Method for increasing system bandwidth through an on-chip address lock register
5,615,283	Dale R. Donchin	Pattern recognition device
5,615,363	Steven M. Jenness	Object oriented computer architecture using directory objects
5,615,382	Vincent G. Gavin, Michael J. Seaman, Neal A. Crook, and Bipin Mistry	Data transfer system for buffering and selectively manipulating the size of data blocks being transferred between a processor and a system bus of a computer system
5,617,283	David B. Krakauer, Kaizad Mistry, Steven Butler, and Hamid Partovi	Self-referencing modulation circuit for CMOS integrated circuit electrostatic discharge protection clamps
5,617,409	Cuneyt M. Ozveren, Hallam G. Murray, Jr., Gregory M. Waters, and Robert J. Simcoe	Flow control with smooth limit setting for multiple virtual circuits
5,619,657	Ram Sudama, David M. Griffin, Brad Johnson, Dexter Sealy, James Shelhamer, and Owen H. Tallman	Method for providing a security facility for a network of management servers utilizing a database of trust relations to verify mutual trust relations between management servers
5,619,662	Simon C. Steely, Jr., David J. Sager, and David B. Fite, Jr.	Memory reference tagging
5,619,710	Robert L. Travis, Jr., Andrew P. Wilson, Neal F. Jacobson, and Michael J. Renzullo	Method and apparatus for object-oriented invocation of a server application by a client application

5,621,678	Michael J. Barnaby and James W. Brissette	Programmable memory controller for power and noise reduction
5,621,734	Bruce E. Mann, Darrell J. Duffy, Anthony G. Lauck, and William D. Strecker	Local area network with server and virtual circuits
5,621,874	Peter Lucas and Jeffrey A. Senn	Three dimensional document representation using strands
5,623,690	Larry G. Palmer and Ricky S. Palmer	Audio/video storage and retrieval for multimedia work-stations by interleaving audio and video data in data file
5,625,802	Hoe T. Cho, Maw Z. Jau, and W. Hugh Durdan	Apparatus and method for adapting a computer system to different architectures
5,625,805	David M. Fenwick, Daniel Wissell, Richard Watson, and Denis Foley	Clock architecture for synchronous system bus which regulates and adjusts clock skew
5,625,822	Bevin R. Brett	Using sorting to do matchup in smart recompilation
5,627,773	Gilbert M. Wolrich, Timothy C. Fischer, and John A. Kowaleski, Jr.	Floating point unit data path alignment
5,627,842	Joseph H. Brown and Dilip K. Bhavsar	Architecture for system-wide standardized intra-module and inter-module fault testing
5,627,981	Michael C. Adler, Steven O. Hobbs, and Paul G. Lowney	Software mechanism for accurately handling exceptions generated by instructions scheduled speculatively due to branch elimination
5,629,840	William R. Hamburgen, John S. Fitch, and Norman P. Jouppi	High powered die with bus bars
5,629,950	Nitin D. Godiwala, Kurt M. Thaller, Jeffrey A. Metzger, and Barry A. Maskas	Fault management scheme for a cache memory
5,630,049	Wayne M. Cardoza, Jeffrey M. Diewald, Jeffrey E. Nelson, Steven D. DiPirro, James R. Goddard, Wendell B. Fisher, Jr., Anne E. McElearney, and Richard Sayde	Method and apparatus for testing software on a computer network
5,630,055	Peter J. Bannon, Ruben W. Castelino, and Chandrasekhara Somanathan	Autonomous pipeline reconfiguration for continuous error correction for fills from tertiary cache or memory
5,630,097	David A. Orbits, Kenneth D. Abramson, and H. Bruce Butts, Jr.	Enhanced cache operation with remapping of pages for optimizing data relocation from addresses causing cache misses
5,630,166	Rodney Gamache, Stuart Farnham, Michael Harvey, William A. Laing, Kathleen Morse, and Michael Uhler	Controlling requests for access to resources made by multiple processors over a shared bus
5,631,908	James B. Saxe	Method and apparatus for generating and implementing smooth schedules for forwarding data flows across cell-based switches
5,633,867	Michael Ben-Nun, Simoni Ben-Michael, Simcha Perl, and Kadangode K. Ramakrishnan	Local memory buffers management for an ATM adapter implementing credit based flow control
5,634,014	William B. Gist and Joseph P. Coyle	Semiconductor process, power supply voltage and temperature compensated integrated system bus termination
5,634,023	Michael C. Adler, Steven O. Hobbs, and Paul G. Lowney	Software mechanism for accurately handling exceptions generated by speculatively scheduled instructions
5,636,355	Kadangode K. Ramakrishnan and Prabuddha Biswas	Disk cache management techniques using non-volatile storage
5,636,366	Scott G. Robinson, Richard L. Sites, and Richard T. Witek	System and method for preserving instruction state-atomicity for translated program
5,638,259	William F. McCarthy, Colin E. Brench, and Daniel M. Snow	Enclosure for electronic modules

5,638,532	Robert C. Frame and Mark J. Foster	Apparatus and method for accessing SMRAM in a computer based upon a processor employing system management mode
5,638,538	Stephen R. Van Doren, Denis J. Foley, and Maurice B. Steinman	Turbotable: apparatus for directing address and commands between multiple consumers on a node coupled to a pipelined system bus
5,644,571	Michael J. Seaman	Apparatus for message filtering in a network using domain class
5,646,581	James O. Pazaris and Richard P. Evans	Low inductance electrical resistor terminator package
5,648,909	Larry L. Biro, Joel J. Grodstein, Jeng-Wei Pan, and Nicholas L. Rethman	Static timing verification in the presence of logically false paths
5,648,959	Nicholas Ilyadis and Richard Graham	Inter-module interconnect for simultaneous use with distributed LAN repeaters and stations
5,649,109	Martin Edward Griesmer, Parayath Gopal Krishnakumar, and David Benson	Apparatus and method for maintaining forwarding information in a bridge or router using multiple free queues having associated free space sizes
5,649,203	Richard Lee Sites	Translating, executing, and re-translating a computer program for finding and translating program code at unknown program addresses
5,650,997	Henry Sho-Che Yang, Anthony G. Lauck, Kadangode K. Ramakrishnan, and William R. Hawe	Method and apparatus for use in a network of the ethernet type, to improve fairness by controlling collision backoff times in the event of channel capture
5,651,111	William M. McKeeman and August G. Reinig	Method and apparatus for producing a software test system using complementary code to resolve external dependencies
5,652,615	Stewart Frederick Bryant and Shaheedur Reza Haque	Precision broadcast of composite programs including secondary program content such as advertisements
5,652,837	Nicholas Allen Warchol and Chester Pawlowski	Mechanism for screening commands issued over a communications bus for selective execution by a processor
5,652,861	David T. Mayo, David W. Hartwell, and Hansel A. Collins	System for interleaving memory modules and banks
5,652,869	Mark A. Herdeg, James A. Wooldridge, Scott G. Robinson, Ronald F. Brender, and Michael V. Iles	System for executing and debugging multiple codes in a multi-architecture environment using jacketing means for jacketing the cross-domain calls
5,652,889	Richard Lee Sites	Alternate execution and interpretation of computer program having code at unknown locations due to transfer instructions having computed destination addresses
5,654,653	Joseph P. Coyle and William B. Gist	Reduced system bus receiver setup time by latching unamplified bus voltage
5,657,239	Joel J. Grodstein, Nicholas L. Rethman, and Jeng-Wei Pan	Timing verification using synchronizers and timing constraints
5,657,426	Keith Waters and Thomas M. Levergood	Method and apparatus for producing audio-visual synthetic speech
5,657,456	William B. Gist and Joseph P. Coyle	Semiconductor process power supply voltage and temperature compensated integrated system bus driver rise and fall time
5,657,471	Richard Lary, Robert Willard, Catharine van Ingen, David Thiel, William Watson, Barry Rubinson, and Verell Boaen	Dual addressing arrangement for a communications interface architecture
5,657,480	Neal F. Jacobson	Method of recording, playback, and re-execution of of concurrently running application program operational commands using global time stamps

5,658,166	Mike Freeman, Stuart Keith Morgan, and Mike Romm	Modular coupler arrangement for use in a building wiring distribution system
5,659,713	Paul M. Goodwin, David A. Tatosian, and Donald Smelser	Memory stream buffer with variable-size prefetch depending on memory interleaving configuration
5,659,739	Clark E. Lubbers, Susan G. Elkington, and Richard F. Lary	Skip list data structure enhancements
5,659,753	Dennis Joseph Murphy and Robert Neil Faiman, Jr.	Interface for symbol table construction in a multi-language optimizing compiler
5,659,775	Alexander Stein and William Grundmann	Topology independent system for state element conversion
5,664,106	Frank Samuel Caccavale	Phase-space surface representation of server computer performance in a computer network
5,664,177	Edward S. Lowry	Data processing system having a data structure with a single, simple primitive
5,664,221	Mark F. Amberg, William K. Miller, Frank M. Nemeth, and Dwayne H. Swanson	System for reconfiguring addresses of SCSI devices via a device address bus independent of the SCSI bus
5,666,415	Charles William Kaufman	Method and apparatus for cryptographic authentication
5,666,519	Peter C. Hayden	Method and apparatus for detecting and executing cross-domain calls in a computer system
5,666,551	David M. Fenwick, Denis J. Foley, Stephen R. Van Doren, David W. Hartwell, Elbert Bloom, and Ricky C. Hetherington	Distributed data bus sequencing for a system bus with separate address and data bus protocols
5,668,951	Rajendra K. Jain, K. K. Ramakrishnan, and Dah-Ming Chiu	Avoiding congestion system for reducing traffic load on selected end systems which utilizing above their allocated fair shares to optimize throughput at intermediate node
5,671,225	Donald F. Hooper, Dave M. Tongel, and Michael B. Evans	Distributed interactive multimedia service system
5,671,406	Clark E. Lubbers and Susan G. Elkington	Data structure enhancements for in-place sorting of a singly linked list
5,675,735	Shawn Gallagher, James Scott Hiscock, Dahai Ding, Scott D'Edwine Lawrence	Method and apparatus for interconnecting network devices in a networking hub
5,675,742	Rajendra K. Jain, K. K. Ramakrishnan, and Dah-Ming Chiu	System for setting congestion avoidance flag at intermediate node to reduce rates of transmission on selected end systems which utilizing above their allocated fair shares
5,675,763	Jeffrey Clifford Mogul	Cache memory system and method for selectively removing stale aliased entries
5,675,800	Wendell Burns Fisher, Jr. and Richard Sayde	Method and apparatus for remotely booting a computer system
5,678,045	Jürgen Bettels	Method and apparatus for multiscript access to entries in a directory
5,680,544	John Edmondson and Scott Taylor	Method for testing an on-chip cache for repair
5,680,584	Mark A. Herdeg and Michael V. Iles	Simulator system for code execution and debugging within a multi-architecture environment
5,680,644	David J. Sager	Low delay means of communicating between systems on different clocks
5,682,489	Jeffrey R. Harrow and Fred P. Messinger	Method and device for monitoring, manipulating, and viewing system information
5,682,551	Chester Walenty Pawlowski, Nicholas Allen Warchol, David Gerard Conroy, and R. Stephen Polzin	System for checking the acceptance of I/O request to an interface using software visible instruction which provides a status signal and performs operations in response thereto
5,684,946	James P. Ellis, Mike Kantrowitz, and Will Sherwood	Apparatus and method for improving the efficiency and quality of functional verification

5,687,310	Paul Stuart Rotker and Randall Dean Hinrichs	System for generating error signal to indicate mismatch in commands and preventing processing data associated with the received commands when mismatch command has been determined
5,687,330	William B. Gist and Joseph P. Coyle	Semiconductor process, power supply and temperature compensated system bus integrated interface architecture with precision receiver
5,689,679	Norman Paul Jouppi	Memory system and method for selective multi-level caching using a cache level code
5,692,159	Mark A. Shand	Configurable digital signal interface using field programmable gate array to reformat data
5,694,312	Gerald J. Brand and Don L. Drinkwater	Uninterruptible power supply with fault tolerance in a high voltage environment
5,694,350	Gilbert M. Wolrich, Timothy C. Fischer, and John A. Kowaleski, Jr.	Rounding adder for floating point processor
5,694,536	Michel Gangnet and Jean-Manuel Van Thong	Method and apparatus for automatic gap closing in computer aided drawing
5,694,579	Rahul Razdan and Gabriel Bischoff	Using pre-analysis and a 2-state optimistic model to reduce computation in transistor circuit simulation
5,695,068	Robert Allison Hart and Richard Harry Plourde	Probe card shipping and handling system
5,696,945	Larry D. Seiler, Robert S. McNamara, Christopher C. Gianos, and Joel J. McCormack	Method for quickly painting and copying shallow pixels on a deep frame buffer
5,696,956	Rahul Razdan, Bill Grundmann, and Michael D. Smith	Dynamically programmable reduced instruction set computer with programmable processor loading on program number field and program number register contents
5,698,818	Colin Edward Brench	Two part closely coupled cross polarized EMI shield
5,701,480	Yoav Raz	Distributed multi-version commitment ordering protocols for guaranteeing serializability during transaction processing
5,701,484	Yeshayahu Artsy	Routing objects on action paths in a distributed computing system
5,701,667	Stephen Michael Birch, Gerard Michel Gavrel, and Zaffar Iqbal Memon	Method of manufacture of an interconnect stress test coupon
5,708,813	Hoe To Cho and Ming Huann Yuan	Programmable interrupt signal router
5,712,858	Nitin Dhiroobhai Godiwala, Andrew Myer Ebert, and Chester Walenty Pawlowski	Test methodology for exceeding tester pin count for an asic device
5,713,009	John Anthony DeRosa, Jr., Benn Lee Schreiber, Peter Chapman Hayden, and Scott Wade Apgar	Method and apparatus for configuring a computer system
5,717,575	Jeffrey P. Copeland and Dennis Robinson	Board mounting system with self guiding interengagement
5,717,729	Russell Iknaian and Richard B. Watson, Jr.	Low skew remote absolute delay regulator chip
5,717,883	David J. Sager	Method and apparatus for parallel execution of computer programs using information providing for reconstruction of a logical sequential program
5,717,921	David Lomet and Betty Salzberg	Concurrency and recovery for index trees with nodal updates using multiple atomic actions
5,720,009	Steven A. Kirk, William Barabash, and William S. Yerazunis	Method of rule execution in an expert system using equivalence classes to group database objects
5,724,033	Michael Burrows	Method for encoding delta values
5,724,513	Michael Ben-Nun, Simoni Ben-Michael, and Moshe De-Leon	Traffic shaping system for asynchronous transfer mode networks

ISSN 0898-901X