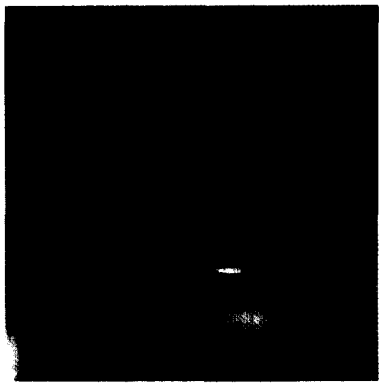


Cache Profiling and the SPEC Benchmarks: A Case Study

Alvin R. Lebeck and David A. Wood
University of Wisconsin – Madison

A vital tool-box component, the CProf cache profiling system lets programmers identify hot spots by providing cache performance information at the source-line and data-structure level.

Cache memories help bridge the cycle-time gap between fast microprocessors and relatively slow main memories. By holding recently referenced regions of memory, caches can reduce the number of cycles the processor must stall while waiting for data. As the disparity between processor and main memory cycle times increases — by 40 percent or more per year— cache performance becomes ever more critical.

Caches only work well, however, for programs that exhibit sufficient locality. Other programs have reference patterns that caches cannot exploit; they spend excessive execution time transferring data between main memory and cache. For example, the SPEC92[1] benchmark tomcatv spends as much as 53 percent of its time waiting for memory on a DECstation 5000/125.

Fortunately, for many programs, small source-code changes — called program transformations — can radically alter memory reference patterns, greatly improving cache performance. Consider the well-known example in Figure 1 of traversing a two-dimensional Fortran array. Since Fortran lays out two-dimensional arrays in column-major order, consecutive elements of a column are stored in consecutive memory locations. Traversing columns in the inner loop (by incrementing the row index) produces a sequential reference pattern and, hence, spatial locality that most caches can exploit. If, instead, the inner loop traverses rows, each inner-loop iteration references a different memory region.

For arrays that are much larger than the cache, the column-traversing version will have much better cache behavior than the row-traversing version. On a DECstation 5000/125, the column-traversing version runs 1.69 times faster than the row-traversing version on an array of single-precision floating-point numbers.

We call this type of analysis a *mental simulation* of the cache behavior. By mentally applying the program reference pattern to the underlying cache organization, we can predict the program's cache performance. This simulation is similar to asymptotic analysis of algorithms (for example, worst-case behavior), which programmers commonly use to study the number of operations executed as a function of input size. When analyzing cache behavior, programmers perform a similar analysis, but they

0018-9162/94/ $4.00 © 1994 IEEE

```
DO 20 K = 1, 100                    DO 20 K = 1, 100
  DO 20 I = 1, 5000                   DO 20 J = 1, 100
    DO 20 J = 1, 100                    DO 20 I = 1, 5000
20    XA(I, J) = 2 * XA(I, J)     20    XA(I, J) = 2 * XA(I, J)
```

(a) **(b)**

Figure 1. Row-major traversal of Fortran array (a), and column traversal (b).

must also have a basic understanding of cache operation (see the next section).

Although asymptotic analysis is effective for certain algorithms, analysis is difficult when applied to large complex programs. Instead, programmers often rely on an execution-time profile to isolate problematic code sections, to which they later apply asymptotic analysis. Unfortunately, traditional execution-time profiling tools (for example, gprof[2]), are generally insufficient to identify cache performance problems. For the Fortran array example (Figure 1), an execution-time profile would identify the procedure or source lines as a bottleneck, but the programmer could easily conclude that the floating-point operations were responsible. We can see, therefore, that programmers would benefit from a profile that focuses specifically on a program's cache behavior.

Our purpose in this article is to introduce a broad audience to cache performance profiling and tuning techniques. Although used sporadically in the supercomputer and multiprocessor communities, these techniques also have broad applicability to programs running on fast uniprocessor workstations. We show that cache profiling, using our CProf cache profiling system, improves program performance by focusing a programmer's attention on problematic code sections and providing insight into appropriate program transformations.

Understanding cache behavior

Caches sit between the (fast) processor and (slow) main memory, holding regions of recently referenced main memory. References satisfied by the cache — called *hits* — proceed at processor speed; those unsatisfied — called *misses* — incur a cache miss penalty to fetch the corresponding data from main memory. Most current processors must wait, or *stall*, until the data arrive.

Caches work because most programs exhibit significant locality. *Temporal locality* exists when a program references the same memory location multiple times in a short period. Caches exploit temporal locality by retaining recently referenced data. *Spatial locality* occurs when the program accesses memory locations close to those it has recently accessed. Caches exploit spatial locality by fetching multiple contiguous words — a *cache block* — whenever a miss occurs.

Caches are characterized by three major parameters: *capacity* (C), *block size* (B), and *associativity* (A). A cache's capacity simply defines the total number of bytes it may contain. The block size determines how many contiguous bytes are fetched on each cache miss. A cache may contain at most C/B blocks at any one time. Associativity refers to the number of unique cache locations where a particular block may reside. If a block can reside in any cache location ($A = C/B$), we call it a *fully associative cache*; if it can reside in exactly one location ($A = 1$), we call it *direct-mapped*; if it can reside in exactly A locations, we call it *A-way set-associative*. (Smith's survey[3] describes cache design in more detail.)

With these three parameters, a programmer can analyze the first-order cache behavior for simple algorithms. Consider the simple example of nested loops where the outer loop iterates L times and the inner loop sequentially acesses an array of N 4-byte integers:

```
for (i = 0; i < L; ++i)
  for (j = 0; j < N; ++j)
    a[j] += 2;
```

If the array size ($4N$) is smaller than the cache capacity (see Figure 2a-b), we expect the number of cache misses to equal the array size divided by the cache block size, $4N/B$ (that is, the number of cache blocks required to hold the entire array). If the array size is larger than the cache capacity (see Figure 2c), the expected number of cache misses is approximately equal to the number of cache blocks required to contain the array times the number of outer-loop iterations ($4NL/B$).

Cache memory terminology

Associativity — The number of unique places in the cache where a particular block may reside.
Block size — The number of contiguous bytes fetched on each cache miss.
Cache hit — A memory reference satisfied by the cache.
Cache miss — A memory reference not satisfied by the cache.
Capacity — The total number of bytes a cache may contain.
Capacity miss — A reference that misses in a fully associative cache with LRU replacement.
Compulsory miss — A reference that misses because it is the first reference to a cache block.
Conflict miss — A reference that hits in a fully associative cache but misses in an A-way set-associative cache.
Direct mapped — A cache in which a block can reside in exactly one place in the cache.
Fully associative — A cache in which a block can reside in any place in the cache ($A = C/B$).
Miss penalty — The time required to fetch data from main memory into the cache on a cache miss.
Set-associative — A cache in which a block can reside in exactly A places in the cache.

Compilers may someday automate this analysis and transform the code to reduce the miss frequency; recent research has produced promising results for restricted problem domains.[4,5] However, for general codes using current commercial compilers, the programmer must manually analyze the programs and manually perform transformations.

To select appropriate program transformations, a programmer must first know what causes poor cache behavior. One approach to understanding why cache misses occur is to classify each miss as one of three disjoint types[6]: *compulsory*, *capacity*, and *conflict*. (Hill and Smith[6] define compulsory, capacity, and conflict misses in terms of miss ratios. When generalizing this concept to individual cache misses, we must introduce anticonflict misses, which miss in a fully associative cache with LRU replacement but hit in an A-way set-associative cache. Anticonflict misses are generally only useful for understanding the rare cases when a set-associative cache performs better than a fully associative cache of the same capacity.)

A compulsory miss is caused by referencing a previously unreferenced cache block. In the small array example (Figure 2b), all misses are compulsory. Eliminating a compulsory miss requires prefetching the data, either by an explicit prefetch operation[5] or by placing more data items in a single cache block. For example, if the integers in our example require only 2 bytes rather than 4, we can cut the misses in half by changing the declaration. However, since compulsory misses usually constitute only a fraction of all cache misses, we do not discuss them further.

A reference that misses in a fully associative cache with LRU replacement is classified as a capacity miss. Capacity misses are caused by referencing more cache blocks than can fit in the cache. In the large array example (Figure 2c), we expect to see many capacity misses. Programmers can reduce capacity misses by restructuring the program to re-reference blocks while they are in cache. For example, it may be possible to modify the loop structure to perform the L outer-loop iterations on a portion of the array that fits in the cache and then move to the next portion of the array. This technique, called *blocking*, is similar to the techniques used to exploit the vector registers in some supercomputers.

A reference that hits in a fully associative cache but misses in an A-way set-associative cache is classified as a *conflict miss*. A conflict miss to block X indicates that block X has been referenced in the recent past, since it is contained in the fully associative cache, but at least A other cache blocks that map to the same cache set have been accessed since the last reference to block X.

Consider the execution of a doubly nested loop on a machine with a direct-mapped cache, where the inner loop sequentially accesses two arrays (for example, dot-product). If the combined array size is smaller than the cache, we might expect only compulsory misses. However, this ideal case occurs only if the two arrays map to different cache sets (Figure 3b). If they overlap, either partially or entirely (Figure 3c), then we will get conflict misses as array elements compete for space in the set. Eliminating conflict misses requires a program transformation that changes either the memory allocation of the two arrays, so that contemporaneous accesses do not compete for the same sets, or that changes the manner in which the arrays are accessed.

Our discussion assumes a cache indexed with virtual addresses. Many systems index their caches with real or physical addresses, making cache behavior strongly dependent on page placement.

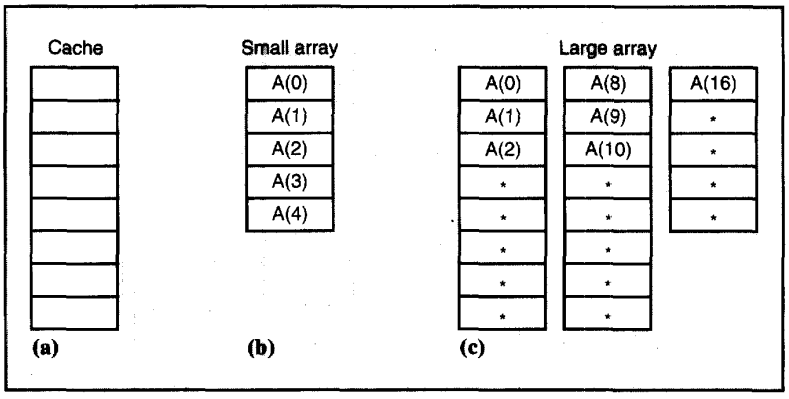


Figure 2. Determining expected cache behavior. Sequentially accessing a small array (b) that fits in the cache (a) should produce M cache misses, where M is the number of cache blocks required to hold the array. Accessing an array that is much larger than the cache (c) should result in ML cache misses, where L is the number of passes over the array.

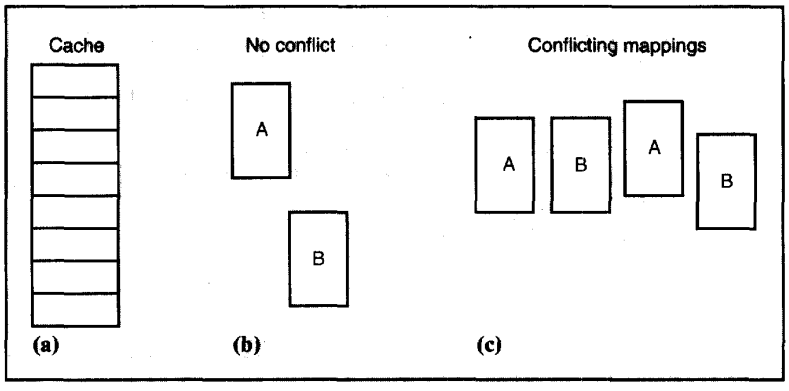


Figure 3. Conflicting cache mappings. The presence of conflict misses indicates a mapping problem: (b) shows how two arrays that fit in the cache (a) with a mapping that will not produce any conflict misses, and (c) shows two mappings that will result in conflict misses.

```
/* old declaration of two arrays */
int val [SIZE];
int key [SIZE];

/* new declaration of */
/* array of structures */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];
```

(a)

```
C old declaration
integer X(N, N)
integer Y(N, N)

C new declaration
integer XY(2*N, N)

C preprocessor macro
C definitions to perform addressing
#define X(i, j) XY((2*i) - 1, N)
#define Y(i, j) XY((2*i), N)
```

(b)

Figure 4. Examples of merging arrays in C (a) and Fortran77 (b).

```
/* old declaration of a twelve */
/* byte structure */
struct ex_struct {
  int val1, val2, val3;
};

/* new declaration of structure */
/* padded to 16-byte block size */
struct ex_struct {
  int val1, val2, val3;
  char pad[4];
};
```

(a)

```
/* original allocation does not */
/* guarantee alignment */
ar = (struct ex_struct *)
  malloc(sizeof(struct ex_struct)*SIZE);

/* new code to guarantee alignment */
/* of structure. */
ar = (struct ex_struct *)
  malloc(sizeof(struct ex_struct)*(SIZE + 1));

ar = ((int) ar + B - 1)/B)*B
```

(b)

Figure 5. Padding (a) and aligning structures (b) in C.

However, many operating systems use page coloring to minimize this effect, thus reducing the performance difference between virtual-indexed and real-indexed caches.[7]

Techniques for improving cache behavior

Program transformations can be classified by the type of cache misses they eliminate. Conflict misses can be reduced by merging arrays, padding and aligning structures, packing structures and arrays, and interchanging loops. The first three techniques change the allocation of data structures, whereas loop interchange modifies the order in which data structures are referenced. Capacity misses can be eliminated by program transformations that reuse data before it is displaced from the cache, such as loop fusion, blocking[4,5] structure and array packing, and loop interchange.

Merging arrays. Some programs contemporaneously reference two (or more) arrays of the same dimension using the same indices. By merging multiple arrays into a single compound array, the programmer increases spatial locality and potentially reduces conflict misses. In the C programming language, this is accomplished by declaring an array of structures rather than two arrays (Figure 4a). This simple transformation can also be performed in Fortran90, which provides structures. Since Fortran77 does not have structures, the programmer can obtain the same effect using complex indexing (Figure 4b).

Padding and aligning structures. Referencing a data structure that spans two cache blocks may incur two misses, even if the structure is smaller than the block size. Padding structures to a multiple of the block size and aligning them on a block boundary can eliminate "misalignment" misses, which generally show up as conflict misses. Padding is easily accomplished in C (Figure 5a) by declaring extra pad fields. Alignment is a little more difficult, since the address of the structure must be a multiple of the cache block size. Statically declared structures generally require compiler support. Dynamically allocated structures can be aligned by the programmer using simple pointer arithmetic (Figure 5b). Some dynamic memory allocators (for example, some versions of *malloc*()) return cache block-aligned memory.

Packing. Packing is the opposite of padding. By packing an array into the smallest space possible, the programmer increases spatial locality, which can reduce conflict and capacity misses. In Figure 6a, the programmer observes that the elements of array *value* are never greater than 255 and, hence, could fit in type *unsigned char*, which requires 8 bits, instead of *unsigned int*, which typically requires 32 bits. For a machine with 16-byte cache blocks, the code in Figure 6b permits 16 elements per block, rather than 4, reducing the maximum number of cache misses by a factor of 4.

Loop fusion. Numeric programs often consist of several operations on the same data, coded as multiple loops over the same arrays. By combining these loops, a programmer increases the program's temporal locality and frequently reduces the number of capacity misses. The examples in Figure 7 combine two doubly nested loops so that all operations are performed on an entire row before moving on to the next. *Loop fusion* is the exact opposite of *loop fission*, a program transformation that splits independent portions of a loop body into separate loops. Loop fission helps an optimizing compiler detect loops that exploit vector hardware on some supercomputers. Because most vector supercomputers do not employ caches, relying instead on high-bandwidth interleaved main memories, some of the transformations described in

this article may be counterproductive for these machines.

Blocking. Blocking is a general technique for restructuring a program to reuse chunks of data that fit in the cache and reduce capacity misses. The SPEC matrix multiply (part of dnasa7, a Fortran77 program) implements a column-blocked algorithm (Figure 8b) that achieves a 2.04 speedup versus a naive implementation (Figure 8a) on a DECstation 5000/125. The algorithm tries to keep four columns of the A matrix in cache for the duration of the outermost loop, ideally getting $N - 1$ hits for each miss. If the matrix is so large that four columns do not fit in the cache, we can use a two-dimensional (row and column) blocked algorithm instead.

```
/* old declaration of an array */
/* of unsigned integers. */
unsigned int values[10000];

/* loop sequencing through values */
for (i = 0; I < 10000; i++)
   values [i] = i % 256;
```

(a)

```
/* new declaration of an array */
/* of unsigned characters. */
/* Valid iff 0 < = value < = 255 */
unsigned char values[10000];

/* loop sequencing through values */
for (i = 0; i < 10000; i++)
   values [i] = i % 256;
```

(b)

Figure 6. Unpacked (a) and packed (b) array structures in C.

CProf cache profiling system

Cache misses *do* result from the complex interaction among algorithm, memory allocation, and cache configuration; when the program is executed, the reality may not match the programmer's expectations. CProf, our cache profiling system, addresses this problem by identifying where cache misses occur and by classifying them as *compulsory*, *capacity*, or *conflict* misses.

Cache- and memory-system profilers differ from the better-known execution-time profilers by focusing on memory-system performance. Memory-system profilers do not obviate execution-time profilers; instead, they provide vital supplementary information to quickly identify memory-system bottlenecks and tune memory-system performance.

Cache- and memory-system profilers differ in the level of detail they present.

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++);
      a[i][j] = 1/b[i][j]*c[i][j];

for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      d[i][j] = a[i][j]+c[i][j];
```

(a)

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
   {
      a[i][j] = 1/b[i][j]*c[i][j];
      d[i][j] = a[i][j]+c[i][j];
   }
```

(b)

Figure 7. Separate (a) and fused (b) loops.

High-level tools, such as MTool,[8] identify procedures or basic blocks that incur large memory overheads. CProf and PFC-Sim,[9] on the other hand, allow more detailed analysis by identifying cache misses at the source-line level. This extra detail is not free; MTool runs much faster than profilers requiring address tracing and full cache simulation. However, full simulation also permits a profiler to identify which data structures are responsible for cache misses and to determine the type of miss — features provided by CProf and MemSpy.[10]

CProf is similar to MemSpy, the difference being the granularity at which source code is annotated and the miss type classification. MemSpy annotates source code at the procedure level and provides two miss types for uniprocessors — compulsory and replacement. CProf provides fine-grain source identification and data structure support, and classifies misses as compulsory, capacity, or conflict.

CProf uses a flexible X Windows interface (see Figure A on p. 20) to present the cache profile in a way that helps the programmer determine the cache performance bottlenecks. The data window lists either source lines or data structures

```
      DO 110 J = 1, M
        DO 110 K = 1, N
          DO 110 I = 1, L
            C(I, K) = C(I, K) + A(I, J) * B(J, K)
110   CONTINUE
```

(a)

```
      DO 110 J = 1, M, 4
        DO 110 K = 1, N
          DO 110 I = 1, L
            C(I, K) = C(I, K) + A(I, J) * B(J, K)
                    + A(I, J + 1) * B(J + 1, K)
                    + A(I, J + 2) * B(J + 2, K)
                    + A(I, J + 3) * B(J + 3, K)
110   CONTINUE
```

(b)

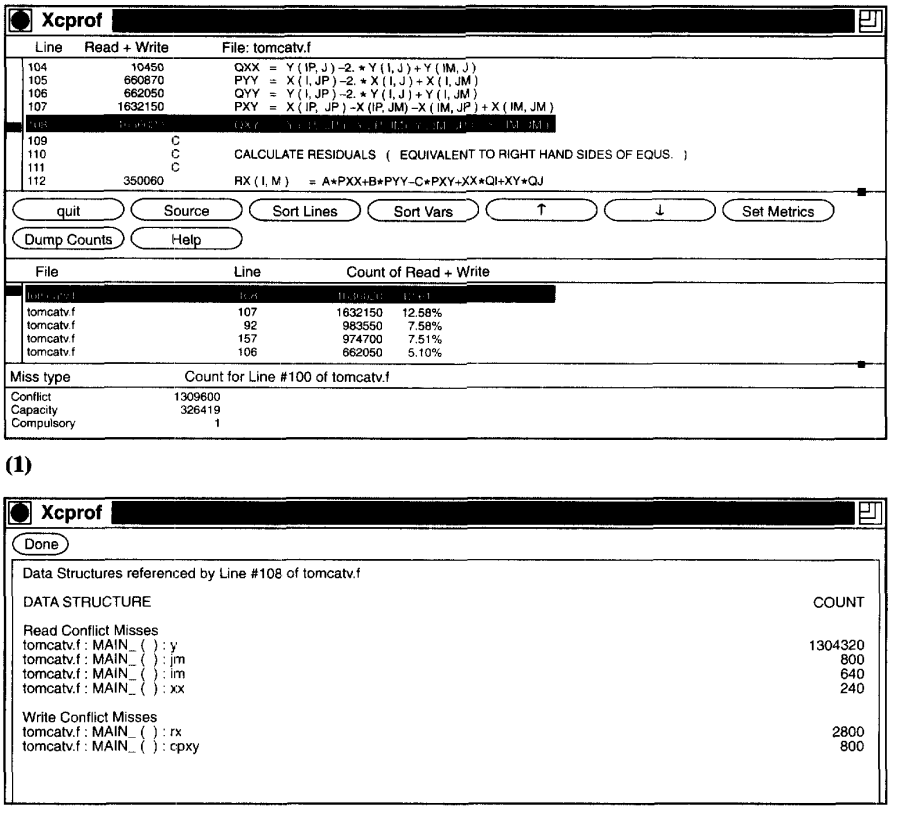
Figure 8. Naive (a) and SPEC column-blocked matrix multiply (b).

CProf user interface

CProf's user interface (Figure A1) is divided into three sections for data presentation and one section for command buttons. The top section is the text window, the middle section is the data window, and the bottom section is the detail window. A particular window's use depends on the selected command button.

The *source* button opens a pull-down menu with an entry for each source file and an additional entry that allows display of a list of source files sorted by the number of cache misses. Selecting one of the files displays the source code in the text window. Each source line is labeled with the number of cache misses generated by that line. We highlight the line with the most cache misses. The up- and down-arrow buttons allow movement within the source file to the line with the next higher or next lower number of misses, respectively. The detail window refines the cache misses for the highlighted line into the miss type. Selecting a miss type causes a window to open that displays the data structures referenced by this source and the corresponding number of cache misses for the miss type selected (Figure A2).

The *sort lines* button displays a list of source lines in the data window, sorted according to the number of cache misses. Each entry contains the file name, the line number, the number of cache misses, and the percent of the total misses. A sorted list of data structures is displayed by the *sort vars* button. Each entry in this list contains the variable name, the count of the number of misses, and the percentage of total misses. Selecting a miss type causes a window to open that displays the source lines that reference this data structure and the corresponding number of cache misses for the miss type selected. The user selects which reference types (*Read, Write, Ifetch*) to display with the *set metrics* button. Finally, the counts displayed in the data window can be written to a file with the *dump counts* button.



(1)

(2)

Figure A. CProf user interface: (1) the primary window and (2) the cross-reference window.

sorted in descending order of importance, allowing quick identification of poor cache behavior. Misses are cross referenced so that a programmer can quickly determine which of several data structures on a source line is responsible for most cache misses.

CProf annotates static and dynamic data structures. Dynamically allocated structures are labeled by concatenating the procedure names on the call stack at the point of allocation.[11] An appended counter value allows unique identification of all dynamically allocated structures.

The *text window* is used to view individual source files, where each line is annotated with the corresponding number of cache misses. The X Windows user interface allows the user to browse within the source file, moving to the line with the next higher or lower number of cache misses. The *detail window* displays the number of each miss type for the currently selected source line or data structure.

Case study: The SPEC benchmarks

Here, we describe a study in which we used CProf and our transformations to tune the cache performance of six programs from the SPEC92 benchmark suite: compress, eqntott, xlisp, tomcatv, spice, and dnasa7. First, we show that we can obtain significant speedups using cache profiling, even for codes that have been extensively tuned using execution-time profilers. Second, we show how we used CProf to gain insight into the cache behavior and determine which transformations were likely to improve performance.

We present performance results in terms of speedup in user execution time on three models of the DECstation 5000: the 5000/240, 5000/125, and 5000/200. (System time accounts for little of the total execution time for most of the programs. Compress is the exception, where system time is relatively high because of the large amount of I/O. In this case, excluding the system time eliminates the bias introduced by the different I/O systems.) Each of these machines has separate 64-Kbyte direct-mapped instruction and data caches, 16-byte blocks, and a write buffer. The 5000/125 and 5000/200 use a 25 MHz MIPS R3000 processor chip. The major difference between the memory systems of the two machines is the cache-miss penalty — 16 processor cycles on the 5000/200 and 34 cycles on the 5000/125, which helps illustrate the importance of cache profiling as cache-miss penalty increases. The 5000/240 uses a 40 MHz MIPS R3000 processor chip and has a 28-cycle miss penalty.

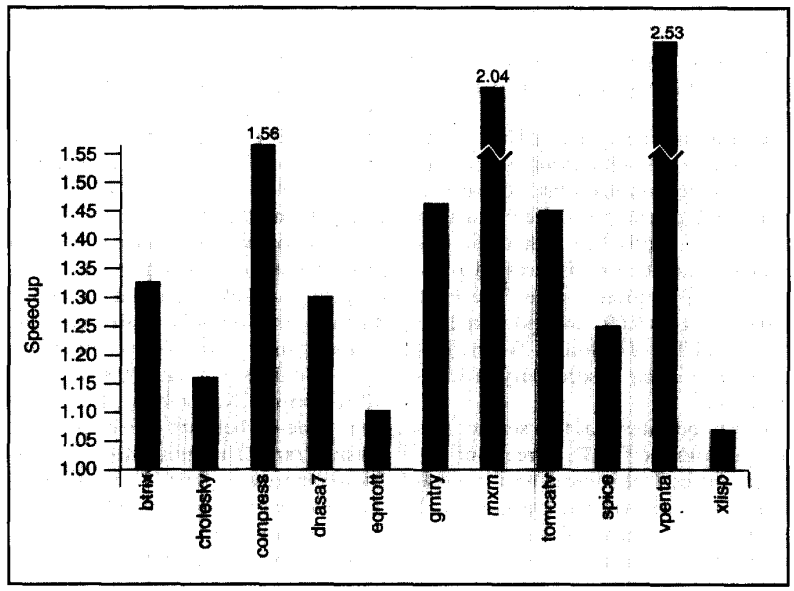


Figure 9. Speedups on a DECstation 5000/125, obtained via cache profiling.

The machines also have secondary differences with significant performance impact. For example, the 5000/240 and 5000/200 have 4-deep write buffers, while the 5000/125 has only a 2-deep write buffer. In addition, the 5000/240 performs sequential prefetch on cache misses, reducing the effective miss penalty for long sequential accesses. While these secondary factors significantly affect execution time, we have not found it necessary to model these factors in CProf's cache simulation.

To reduce experimental error, we averaged the execution time over five runs. The programs were compiled at optimization level –O3 using the MIPS Version 2.1 C and F77 compilers. Spice was the one exception, compiled at optimization level –O2 per the SPEC make file. While run times are all reported with full optimization, we profiled most of the programs at optimization level –O1, with full symbolic debugging (–g). Cache profiling at high optimization levels suffers from the same difficulties as debugging (that is, incorrect line numbers), since CProf uses the same symbol table information.

Table 1 shows the applications that benefited from the various restructuring techniques. The benchmark program dnasa7 consists of seven numerical kernels; we broke out five kernels with poor cache performance and analyzed them separately.

Table 2 on the next page and Figure 9 above present execution time results for the six benchmarks. The full programs execute as much as 90 percent faster when modified to improve cache behavior. Breaking out the kernels in dnasa7 shows even more striking results, with speedups as much as 3.46 for vpenta on

Table 1. Program restructuring techniques that improve the cache behavior of each program studied.

	Restructuring Technique					
Program	Merging Arrays	Loop Fusion	Loop Interchange	Padding and Aligning	Packing	Blocking
btrix*	•	•	•			
cholesky*			•			
compress	•					•
eqntott					•	
gmtry*			•			
mxm*						•
spice	•					
tomcatv	•	•				
vpenta*	•	•	•			
xlisp				•		

* dnasa7

the 5000/240, 2.53 on the 5000/125, and 2.14 on the 5000/200.

Below we discuss our experience cache profiling and modifying each program.

Compress. The compress Unix utility implements the well-known Lempel-Ziv data compression algorithm. For each input character, compress searches a hash table for a prefix key. When the key matches, another array is accessed to obtain the appropriate value. The hash table is large (69,001 entries) to reduce the probability of collisions. When a collision does occur, a secondary probe is initiated.

CProf indicates that two source lines are responsible for 71 percent of the cache misses. One source line, the initial probe into the hash table, accounts for 21 percent of the cache misses. The other source line performs the secondary probe operation when there is a collision; it accounts for 50 percent of the misses. CProf also shows that most of the misses are capacity misses. We can eliminate capacity misses by processing data in portions that fit in the cache.

Applying this insight to compress, we reduced the hash table size from 69,001 to 5,003, small enough to fit in the data cache. This change results in speedups of 1.92 on a 5000/240, 1.56 on a 5000/125, and 1.30 on a 5000/200. However, this modification actually changes the program output, since the compression ratio (original file size/compressed file size) is related to the size of the hash table. The output is still a compatible compressed file, but it does not match the standard SPEC output. Nonetheless, there is a clear trade-off between speed and compression ratio. The unoptimized version has a compression ratio of 2.13, whereas the optimized version's is 1.77.

We also tried to improve the cache performance of compress without changing the compression ratio. Although compress has a large number of capacity misses, conflict misses account for 13 percent of the misses to the key array and 19 percent of the misses to the value array. CProf's X Windows interface allowed us to quickly determine that the array index is the same for both arrays. Although separate arrays reduce the total space requirements (the key is a C integer and the value is a short; alignment restrictions in C require padding if these are combined into an array of structures), the price is poor spatial locality. After referencing a key, compress is likely to reference the corresponding value, which resides in the other array and in a different cache block (see Figure 10a).

Merging the two arrays into a single array of structures places the key and value in the same cache block (see Figure 10b), improving spatial locality. With this modification, accesses to the value always hit in the cache (assuming proper alignment), reducing the number of conflict misses and providing speedups of 1.07 on the 5000/240, 1.05 on the 5000/125, and 1.02 on the 5000/200.

Table 2. Execution time speedup resulting from cache profiling. (The original and tuned times for dnasa7 include the SPEC version of matrix multiply or mxm.)

	Machine						
	5000/125		5000/200		5000/240		
Program	Seconds	Speedup	Seconds	Speedup	Seconds	Speedup	Modification
compress	7.70		5.98		5.56		Original
	7.34	1.05	5.84	1.02	5.22	1.07	Merged key and value arrays
	4.94	1.56	4.60	1.30	2.90	1.92	Reduced hash table size
dnasa7	**1228.22**		**904.60**		**796.60**		**Original**
	945.18	**1.30**	**727.84**	**1.24**	**527.24**	**1.51**	**Tuned kernels**
btrix	144.06		114.50		82.52		Original
	109.50	1.32	89.92	1.27	55.94	1.48	Loop interchange and loop fusion
cholesky	**188.90**		**141.14**		**97.14**		**Original**
	162.16	**1.16**	**124.94**	**1.13**	**73.66**	**1.32**	**Loop interchange**
gmtry	177.06		141.98		128.42		Original
	119.78	1.48	95.82	1.48	50.92	2.52	Loop interchange
mxm	**248.44**		**184.56**		**91.36**		**Naive**
	122.06	**2.04**	**106.02**	**1.74**	**66.08**	**1.38**	**SPEC column blocked**
vpenta	264.78		169.86		203.80		Original
	126.38	2.10	91.80	1.85	69.60	2.93	Merged arrays and loop interchange
	104.54	2.53	79.42	2.14	58.88	3.46	+ loop fusion
eqntott	**67.56**		**58.70**		**39.96**		**Original**
	60.98	**1.11**	**55.40**	**1.06**	**38.92**	**1.03**	**Changed short to character**
spice	2242.10		1762.34		1557.90		Original
	1781.72	1.26	1406.04	1.25	1163.42	1.34	Merged pointer and number
tomcatv	**221.20**		**161.20**		**137.30**		**Original**
	167.24	**1.32**	**134.38**	**1.20**	**91.40**	**1.50**	**Merged arrays X and Y**
	150.88	**1.47**	**126.36**	**1.28**	**86.08**	**1.60**	**+ loop fusion**
xlisp	385.24		286.56		205.72		Original
	361.96	1.06	277.18	1.03	190.30	1.08	Padded node to 16 bytes

Eqntott. The SPEC benchmark eqntott is a CAD tool that converts Boolean equations into their equivalent truth tables. Execution-time profiling shows that eqntott spends 95 percent of its time in the quick-sort routine.[12] CProf further reveals that most of this time is spent moving the sort keys from memory into the cache; more than 90 percent of the misses are generated in one comparison routine. The offending routine examines two arrays and generates mostly capacity misses, indicating that we either need to re-reference blocks while they are in the cache or bring in fewer blocks. CProf indicates that most of these capacity misses are due to fetching BIT structures dynamically allocated at line No. 44 in pterm.c. The BIT data type is a 16-bit integer (type short in C), and inspection of the source code shows that BIT data types only take on values in the set [0, 1, 2]. Changing the type definition from 16-bit integer to 8-bit integer (short to char) reduces the number of misses in this routine by half. The speedup in execution time is 1.03 on a 5000/240, 1.11 on a 5000/125, and 1.06 on a 5000/200. The prefetch capabilities of the 5000/240 exploit the sequential accesses of the compare routine, reducing the benefit of our modification.

In eqntott, the integer values actually represent the symbolic values zero, one, and dash. With the use of enumerated types, a compiler could potentially allocate as few as two bits per array element, resulting in one-eighth the number of cache misses. The trade-off, however, between fewer cache misses and the time to unpack the data is implementation dependent.

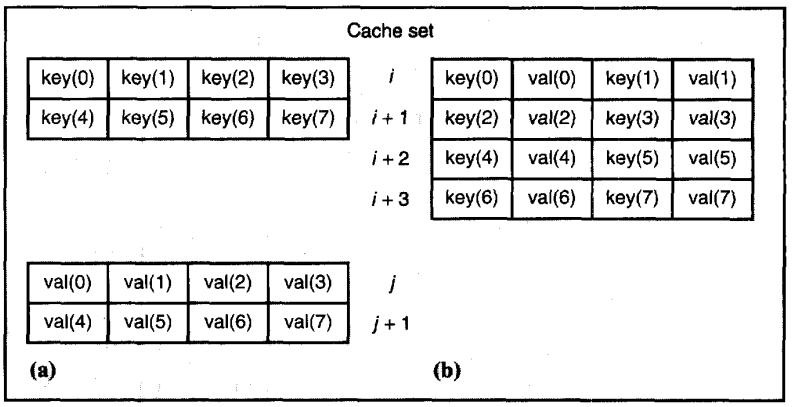


Figure 10. Cache mappings for compress. The initial allocation strategy for the key and value arrays (a) resulted in as many as two cache misses for each successful hash table probe. Merging the two arrays into an array of structures (b) effectively interleaves the elements of the two arrays and results in only one cache miss per successful probe.

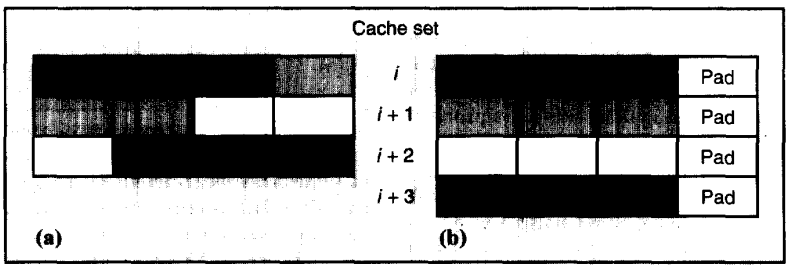


Figure 11. Cache mappings for xlisp node structures. Each pattern corresponds to a different node structure, while *pad* indicates wasted storage. The initial allocation strategy (a) resulted in two cache misses for half of the nodes not in the cache. Padding the structures to equal a cache block size and alignment on cache block boundaries (b) reduces this to only one cache miss per node not resident in the cache.

Xlisp. The SPEC benchmark xlisp is a small lisp interpreter solving the nine queens problem. To reduce computation requirements during profiling, we profiled xlisp solving the six queens problem; however, the speedup results in Table 2 are for the standard nine queens input. Programmers should be aware that cache behavior is sensitive to the input data. Programs may exhibit good cache behavior with smaller input sizes and poor behavior for larger inputs. In this case, the results obtained from the smaller input data were sufficient to achieve reasonable speedups with the larger input.

CProf shows that approximately 40 percent of the cache misses (mostly the conflict type) occur during mark-and-sweep garbage collection. During this phase, the program first traverses the reachable nodes and marks them accessible, and then sweeps sequentially through the memory segment placing unmarked nodes on the free list. Mark-and-sweep garbage collection has inherently poor locality, and an alternate algorithm would provide better cache behavior.

CProf also shows that 19 percent of the cache misses are generated by the single source line that checks the flag (used to mark accessibility) during the sweep. Since conflict misses dominate, we first improved the spatial locality of the sweep routine by separating the flags from the rest of the node structure. With the flags in a single array, the sequential sweep exhibited excellent spatial locality. For every miss, the next 15 references hit, eliminating most of the cache misses in the sweep routine. Unfortunately, the change also increased the number of misses in the mark routine, which must first fetch a node, then the corresponding flag. This modification increased spatial locality in the sweep at the expense of spatial locality during the mark, resulting in a negligible change in performance.

Returning to CProf, the node structures allocated on line No. 540 of xldmem.c incur a large number of conflict misses. Inspection of the source reveals that each node structure occupies 12 bytes, or three-fourths of a 16-byte cache block. Consequently, only half the nodes reside entirely within a single cache block (see Figure 11). The remaining nodes reside in two contiguous cache blocks, potentially causing two cache misses rather than one. By explicitly padding the original node structure to 16 bytes (the cache

block size) and ensuring alignment on cache-block boundaries, we obtained a 1.08 speedup on the 5000/240, 1.06 on the 5000/125, and 1.03 on the 5000/200.

Padding data structures without guaranteeing alignment can be worse than not padding them at all. In this example, we might end up with *all* nodes generating two misses. Similarly, while many memory allocators (for example, the Ultrix malloc() routine) return cache-block-aligned memory, xlisp preallocates large chunks and manages them itself, bypassing the alignment performed within the allocator. Application-specific memory managers certainly have a role, but programmers should remember the impact of padding and alignment on cache performance.

Padding data structures also wastes memory space; the xlisp node structures use only 10 bytes of information. Explicit padding increases the allocated size from the 12 bytes required by C language semantics to 16 bytes — a 33 percent increase in storage. Increasing byte allocation could adversely affect virtual memory performance for larger programs, although that was not a problem in this case.[12]

```
      for LL

          /* FORWARD WAVE */
loop 1.     for j
                for i
                    X, Y RX, RY, AA, DD
loop 2.     for j
                for i
                    RX, RY
loop 3.     for j
                for i
                    AA, DD RX, RY, D

          /* BACKWARD WAVE */
loop 4.     for j
                for i
                    RX, RY, AA, D

          /* FORWARD WAVE */
loop 5.     for j
                for i
                    X, Y RX, RY

      endfor
```

(a)

```
      for LL

          /* FORWARD WAVE */
loop 1.     for j
                for i
                    X, Y RX, RY
                for i
                    X, Y RX, RY, AA, DD
                for i
                    RX, RY
                for i
                    AA, DD RX, RY, D

          /* BACKWARD WAVE */
loop 2.     for j
                for i
                    RX, RY, AA, D
      end for
```

(b)

Figure 12. Original tomcatv pseudocode (a), and loop-fused tomcatv (b). The original tomcatv algorithm contains several loops within a forward wave. Although the same arrays are referenced in consecutive loops, the data accessed in the beginning of the loop is displaced by data referenced at the end of the previous loop. The loop-fused version of tomcatv performs all forward-wave operations on one row of the arrays. This results in speedups of 1.60, 1.47, and 1.28 on the DECstation 5000/240, 5000/125, and 5000/200, respectively.

Tomcatv. Tomcatv is a Fortran77 mesh generation program that uses seven two-dimensional data arrays, each of which requires approximately 0.5 Mbyte. The algorithm (see Figure 12a) consists of a forward pass in which two arrays are read and the other five written (loops 1, 2, 3), a backward pass (loop 4) over two arrays to calculate errors, and finally another forward pass (loop 5) to add these errors.

Since the arrays are much larger than the cache and are sequentially accessed, we expect to see a large number of capacity misses. However, CProf shows that read accesses to arrays X and Y during the first loop of the initial forward pass are generating a large number of conflict misses. The two arrays are always referenced with the same indices. Hence, to improve spatial locality, we merged them, placing elements $X(I,J)$ and $Y(I,J)$ in the same cache block. This modification results in speedups of 1.50 on the 5000/240, 1.32 on the 5000/125, and 1.20 on the 5000/200.

Running CProf on the modified tomcatv, we find that capacity misses to the RX and RY arrays now dominate. As Figure 12a shows, the forward pass is actually composed of several loops. Loop 1 initially references six arrays, including writing RX and RY, followed by loop 2 that computes the maximum values of the RX and RY arrays, and a final pass (loop 3) over the RX and RY arrays to adjust the values. In addition to these disjoint forward pass loops, there is the forward pass (loop 5) to add the errors to the X and Y arrays after the backward pass (loop 4) over the RX and RY arrays. The RX and RY arrays are referenced in the same order in each loop of the forward pass (loops 1, 2, 3). However, each array is 0.5 Mbyte in size, much larger than the 64-Kbyte data cache. Therefore, the elements referenced at the start of one loop are not in the cache when the next loop starts.

The solution is to improve temporal locality by restructuring the program so that all allowable operations are performed on an element when it is cache resident. Transforming the program via loop fusion (Figure 12b) merges these loops so that the program contains only one forward and one backward loop. We cannot perform the operations of both the forward pass and backward pass in the same loop because of data dependencies. We folded the addition of error corrections into the forward pass. Loop fusion, in addition to array merging, produced a speedup of 1.60 on the 5000/240, 1.47 on the 5000/125, and 1.28 on the 5000/200. These speedups are not as high as we expected because of an increase in the number of conflict misses and a slight increase in the number of instructions executed.

Spice. Spice (spice2g6) is an analog circuit simulator written in Fortran. The primary data structure is a sparse matrix, which is implemented by several arrays. In particular, there are separate arrays for row pointers, row numbers, column pointers, column numbers, and values. CProf shows that two source lines accessing the row pointer and number ar-

```
      DO 8 I = 1, MATDIM
        RMATRX(I, I) = 1.D0 / RMATRX(I, I)
        DO 8 J = I + 1, MATDIM
          RMATRX(J, I) = RMATRX(J, I) *RMATRX(I, I)
          DO 8 K = I + 1, MATDIM
            RMATRX(J, K) = RMATRX(J, K)
                  - RMATRX(J, I) * RMATRX(I, K)
    8 CONTINUE
```

(a)

```
      DO 8 I = 1, MATDIM
        RMATRX(I, I) = 1.D0 / RMATRX(I, I)
        DO 81 J = I + 1, MATDIM
          RMATRX(J, I) = RMATRX(J, I) * RMATRX(I, I)
   81   CONTINUE
        DO 8 K = I + 1, MATDIM
          DO 8 K = I + 1, MATDIM
            RMATRX(J, K) = RMATRX(J, K)
                  - RMATRX(J, I) * RMATRX(I, K)
    8 CONTINUE
```

(b)

Figure 13. Gaussian elimination loops: (a) original; (b) interchanged.

rays cause 34 percent of the cache misses, with an additional 12 percent from another two source lines accessing the column pointer and number arrays. Each source line pair is contained in a small loop that locates an element (I, J) in the sparse matrix. CProf shows that most misses caused by these source lines are conflict misses, indicating a mapping problem. Again, the X Windows interface of CProf allows us to quickly determine that the row (column) pointer and row (column) number arrays are nearly always accessed with the same index. Merging the pointer and number arrays to improve spatial locality results in a speedup of 1.34 on the 5000/240, 1.26 on the 5000/125, and 1.25 on the 5000/200.

Dnasa7: The NASA kernels

Dnasa7 is a collection of seven floating-point intensive kernels also known as the NAS kernels: vpenta, cholesky, btrix, fft, gmtry, mxm and emit. Each kernel initializes its arrays, copies them to working arrays, and then calls the application routine. We discuss the kernels separately, to better describe the cache optimizations. We did not study emit, a vortex generation code, or fft, a fast Fourier transform code. Emit has a low miss ratio on a 64-Kbyte data cache (0.8 percent), and shuffling ffts have inherently poor cache performance. The speedup we obtained for the entire collection of kernels is 1.51 on the 5000/240, 1.30 on the 5000/125, and 1.24 on the 5000/200.

Vpenta. The vpenta kernel simultaneously inverts three pentadiagonals, a routine commonly used to solve systems of partial differential equations. CProf first finds that the miss ratio is a startling 36 percent, mostly due to conflict misses. Using CProf to identify the mapping problems, we discovered two nested loops responsible for more than 90 percent of the cache misses. One loop accesses three arrays, while the other accesses eight arrays. We can eliminate conflict misses by changing the data structure allocation or the order in which structures are accessed. Source code inspection reveals that both techniques can be applied. We discovered the loops could be interchanged to traverse the arrays in column order and identified three opportunities for array merging. These modifications result in speedups of 2.93 on a 5000/240, 2.10 on a 5000/125, and 1.85 on the 5000/200. The original code runs slower on the 5000/240 than on the 5000/200, despite the 60-percent faster processor cycle time. This is apparently due to the higher miss penalty — the two machines use the same DRAMs, but the 240 incurs approximately 100 nanoseconds additional delay, due to an asynchronous interface. Loop interchange not only increases spatial locality, but results in a sequential access pattern that the 240's prefetch logic can exploit. The 5000/240 has a speedup of 1.3 versus the 5000/200 on the modified code.

As with tomcatv, running CProf on the modified version of vpenta shows that capacity misses now dominate. Fusing loops, which eliminates multiple passes over the same arrays to improve temporal locality, results in speedups over the original version of 3.46, 2.53, 2.14 on the 5000/240, 5000/125, and 5000/200, respectively.

Cholesky. Cholesky performs cholesky decomposition and substitution. CProf reveals numerous capacity misses in two nested loops. Source code inspection identifies an array traversed in row-major, rather than column-major, order. Statically transposing the array (effectively performing loop interchange but with much simpler code modification) results in speedups of 1.32 on the 5000/240, 1.16 on the 5000/125, and 1.13 on the 5000/200. Blocking can also be applied to cholesky,[4] but we chose to apply a much simpler transformation.

Btrix. Btrix is a tridiagonal solver. CProf shows that most misses are again capacity misses that occur in two nested loops. As always, we first checked the array reference order and immediately noticed that one array is traversed in row order. We also observed that statically transposing this array would allow fusion of six different loops. We were able to apply several transformations after a single run of CProf. We obtain a speedup of 1.48 on the 5000/240, 1.32 on the 5000/125, and 1.27 on the 5000/200.

Gmtry. Gmtry is a kernel dominated by a Gaussian elimination routine (see Figure 13). CProf finds that 99 percent of the misses, mostly capacity, occur in the Gaussian elimination loop; inspection shows that the rmatrx is traversed in row order. Interchanging the loops, which is trivial in this case, results in a speedup of 2.52 on the 5000/240 and 1.48 on the 5000/200 and 5000/125.

Mxm. Mxm is a matrix-matrix multiply routine. The naive matrix multiply algorithm is a well-known "cache buster" because there is little data reuse between

loop iterations. Instead of this algorithm, the SPEC mxm implementation uses a column-blocked implementation (described above) that reuses the same four columns throughout the two innermost loops. Cache performance improvement was not the original rationale for blocking mxm; instead, the intent was to let vectorizing compilers more effectively reuse the contents of vector registers in Cray supercomputers. In this case, the same transformation improves performance for both vector registers and caches.

The standard SPEC column-blocked algorithm achieves a speedup of 1.38 versus the naive algorithm on a 5000/240, 2.04 on the 5000/125, and 1.74 on a 5000/200. For larger matrices, a 2D row- and column-blocked algorithm would perform better, but for the standard SPEC input size, the extra overhead decreases performance.

As processor cycle times continue to increase faster than main memory cycle times, memory hierarchy performance becomes increasingly important. Programmers can mentally simulate cache behavior to help select algorithms with good cache performance.

Actual cache performance, unfortunately, does not always match the programmer's expectations, and many programs are too complex for the interactions among memory reference patterns, data allocation, and cache organization to be fully analyzed. In these cases, a tool like CProf becomes an important element in a programmer's tool box.

CProf offers cache performance information at the source line and data structure level, which allows a programmer to identify hot spots. By classifying cache misses as compulsory, capacity, or conflict, CProf lets programmers select appropriate program transformations that improve a program's spatial or temporal locality, leading to better overall performance. ■

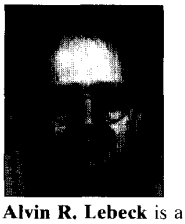
Acknowledgments

We thank Karen Miller, Alain Kagi, and Chris Maguire for work on an earlier version of CProf. Kagi and Scott Kempf found several bugs in the latest version. James Larus provided a great deal of support for QPT, used to generate input for CProf, and suggested reducing the hash table size in compress. Mitali Lebeck, Larus, and Mark Hill provided helpful comments and suggestions on early drafts.

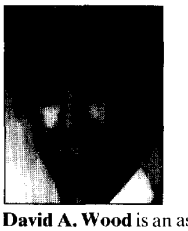
This work is supported in part by National Science Foundation Presidential Young Investigator Awards CCR-9157366 and MIPS-8957278, National Science Foundation Grants CDA-8920777, CCR-9101035, and MIP-922-5097, and donations from AT&T, Bell Laboratories, Digital Equipment Corporation, Xerox Corporation, and the Graduate School of the University of Wisconsin.

References

1. *SPEC Newsletter*, Standard Performance Evaluation Corp., Fairfax, Va., Dec. 1991.
2. S.L. Graham, P.B. Kessler and M.K. McKusick, "An Execution Profiler for Modular Programs," *Software Practice and Experience*, Vol. 13, 1983, pp. 671-685.
3. A.J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, Sept. 1982, pp. 473-530.
4. M.S. Lam, E.E. Rothberg, and M.E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. ASPLOS 4*, 1991, ACM, New York, pp. 63-74.
5. A. Porterfield, "Software Methods for Improvement of Cache Performance on Supercomputer Applications," PhD thesis, Dept. of Computer Sci., Rice Univ., 1989.
6. M.D. Hill and A.J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Trans. Computers*, Vol. 38, No. 12, Dec. 1989, pp. 1,612-1,630.
7. R.E. Kessler and M.D. Hill, "Page Placement Algorithms for Large Real-Index Caches," *ACM Trans. Computer Systems*, Vol. 10, No. 4, Nov. 1992, pp. 338-359.
8. A.J. Goldberg and J. Hennessy, "Performance Debugging Shared-Memory Multiprocessor Programs with MTool," *Proc. Supercomputing 91*, IEEE CS Press, Los Alamitos, Calif., Order No. 2158, 1991, pp. 481-490.
9. D. Callahan, K. Kennedy and A. Porterfield, "Analyzing and Visualizing Performance of Memory Hierarchies," *Instrumentation for Visualization*, ACM Press, New York, 1990.
10. A. Gupta, M. Martonosi, and T. Anderson, "MemSpy: Analyzing Memory-System Bottlenecks in Programs," *Performance Evaluation Rev.*, Vol. 20, No. 1, June 1992, pp. 1-2.
11. B. Zorn and P.N. Hilfinger, "A Memory Allocation Profiler for C and Lisp," *Proc. Summer 1988 Usenix Conf.*, Usenix Assoc., Berkeley, Calif., 1988.
12. D.N. Pnevmatikatos and M.D. Hill, "Cache Performance of the Integer SPEC Benchmarks on a RISC," *ACM SIGArch Computer Architecture News*, Vol. 18, No. 2, June 1990, pp. 53-68.



Alvin R. Lebeck is a PhD candidate in the Computer Sciences Department at the University of Wisconsin – Madison, where he received his BS in electrical engineering in 1989 and his MS in computer science in 1991. His research interests include memory-system design for uniprocessors and multiprocessors, efficient simulation of memory systems, and synchronization techniques for multiprocessors. He is a member of the IEEE, the IEEE Computer Society, and ACM.



David A. Wood is an assistant professor in the Computer Sciences and Electrical and Computer Engineering departments at the University of Wisconsin – Madison. His research interests range from VLSI design to operating systems, and focus on design and evalution of computer architectures with an emphasis on memory systems for shared-memory multiprocessors.

Wood received his BS in electrical engineering and computer science at the University of California, Berkeley, in 1981 and his PhD in computer science there in 1990. He is a coleader of the National Science Foundation-sponsored Wisconsin Wind Tunnel Project. He won a 1991 NSF's Presidential Young Investigator award, and is a member of ACM, the IEEE, and the IEEE Computer Society.

Readers can contact the authors at the Computer Sciences Dept., University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706. Lebeck's e-mail address is alvy@cs.wisc.edu, and Wood's is david@cs.wisc.edu.