Cache Profiling and the SPEC Benchmarks: A Case Study

Alvin R. Lebeck
David A. Wood

Computer Sciences Department
University of Wisconsin - Madison
1210 West Dayton Street
Madison, WI. 53706
(608) 262-6617
alvy@cs.wisc.edu

To appear IEEE Computer

Abstract

As VLSI technology improvements continue to widen the gap between processor and main memory cycle times, cache performance becomes increasingly important to overall system performance. Cache memories help alleviate the cycle time disparity, but only for programs that exhibit sufficient spatial and temporal locality. Programs with unruly access patterns spend much of their time transferring data to and from the cache. To fully exploit the performance potential of fast processors, programmers must explicitly consider cache behavior, restructuring their codes to increase locality. As these fast processors proliferate, techniques for improving cache performance must move beyond the supercomputer and multiprocessor communities and into the mainstream of computing.

In this paper, we examine some of the techniques that programmers can use to improve cache performance. We show how to use CPROF, a cache profiler, to identify cache performance bottlenecks and gain insight into their origin. This insight helps programmers understand which of the well-known program transformations are likely to improve cache performance. Using CPROF and a "cookbook" of simple transformations, we show how to tune the cache performance of six of the SPEC92 benchmarks. By restructuring the source code, we greatly improve cache behavior and achieve execution time speedups ranging from 1.02 to 3.46.

Keywords:

Performance, Cache Memories, Cache Profile, Program Behavior, Performance Tuning

Introduction

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 waiting for data. As the disparity between processor and main memory cycle times increases—by

40% per year or more— cache performance becomes ever more critical.

Unfortunately, caches only work well for programs that exhibit sufficient locality. Other programs have reference patterns that caches cannot exploit, and spend much of their execution time transferring data between main memory and cache. For example, the SPEC92 [8] benchmark `tomcatv` spends 53% of its time waiting for memory on a DECstation 5000/125.

Fortunately, for many programs small changes in the source code can radically alter their memory reference pattern, greatly improving cache performance. Consider the well-known example 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 region of memory.

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

Row-Major Traversal

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

Column-Major Traversal

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 mental simulation is similar to asymptotic analysis of algorithms (e.g., worst-case behavior), that 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 must also have a basic understanding of cache operation (see the following section).

Although asymptotic analysis is effective for certain algorithms, analyzing large complex programs is very difficult. Instead, programmers often rely on an execution-time profile to isolate problematic code sections, and then apply asymptotic analysis only on those sections. Unfortunately, traditional execution-time profiling tools, e.g.,

gprof [3], are generally insufficient to identify cache performance problems. For the example above, 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. Instead, programmers would benefit from a profile that focuses specifically on a program's cache behavior, identifying problematic code sections and data structures. Cache profiles can also help provide insight into the cause of cache misses, which can help a programmer determine appropriate program transformations to improve performance.

The purpose of this article is to introduce a broad audience to the techniques of cache performance profiling and tuning. Although these techniques have been used sporadically in the supercomputer and multiprocessor communities, they also have broad applicability to programs running on fast uniprocessor workstations. We show that cache profiling—using the CPROF cache profiling system—is an effective means of improving program performance by focusing a programmer's attention on problematic code sections and providing insight into the type of program transformation to apply.

In the next section, we review how to reason about cache behavior and show how knowing the cause of a cache miss helps provide insight into how to eliminate it. We then present a ''cookbook'' of simple program transformation techniques for improving program cache behavior, including array merging, padding and aligning structures, structure and array packing, loop interchange, loop fusion, and blocking. We then briefly describe the CPROF cache profiling system and its X-windows based user interface. Then we present a case study where we used CPROF to tune the cache performance of six programs from the SPEC92 benchmark suite: `compress`, `dnasa7`, `eqntott`, `spice`, `tomcatv`, and `xlisp`. We show how CPROF identified the source lines and data structures that exhibit poor cache behavior, and how CPROF helped provide the insight necessary to select the appropriate program transformation. Execution time speedups for these programs range from 1.02 to 3.46, depending on the machine's memory system.

Understanding Cache Behavior: A brief review

To reason about a program's cache behavior, programmers must first recall the basic operation of cache memories. 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 proces-

sors 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.

Cache Memory Terminology

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

Caches are characterized by three major parameters: Capacity (**C**), Block Size (**B**), and Associativity (**A**). A cache's capacity (**C**) simply defines the total number of bytes it may contain. The block size (**B**) 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 (**A**) refers to the number of unique locations in the cache a particular block may reside in. If a block can reside in any location in the cache ($A=C/B$) 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 [11] provides a more detailed description of cache design.)

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 accesses an array of N 4-byte integers.

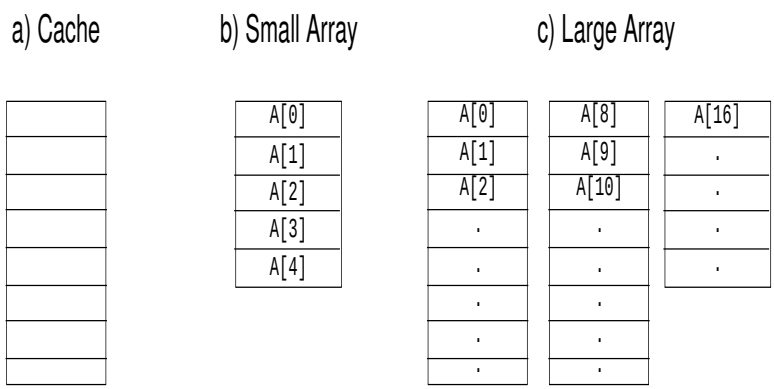


Figure 1: Determining Expected Cache Behavior

Sequentially accessing an array that fits in cache (Figure 1b) 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 (Figure 1c) should result ML cache misses, where L is the number of passes over the array.

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

If the size of the array (4N) is smaller than the cache capacity (see Figure 1b), we expect the number of cache misses to equal the size of the array divided by the cache block size, 4N/B (i.e., the number of cache blocks required to hold the entire array). If the size of the array is larger than the cache capacity (see Figure 1c), 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).

Someday compilers may automate this analysis and transform the code to reduce the miss frequency; recent research has produced promising results for restricted problem domains [7, 10]. However, for general codes using current commercial compilers, the programmer must manually analyze the programs and perform transformations by hand.

To select appropriate program transformations, a programmer must first obtain insight into the cause of poor cache behavior. One approach to understanding the cause of cache misses, is to classify each miss into one of three

disjoint types [5]: *compulsory*, *capacity*, *conflict*.[1] A compulsory miss is caused by referencing a previously unreferenced cache block. In the small array example above (see Figure 1b), all misses are compulsory. Eliminating a compulsory miss requires prefetching the data, either by an explicit prefetch operation [10] 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 a small fraction of all cache misses we do not discuss them further in this article.

A reference that is not a compulsory miss but 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 above (see Figure 1c), 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 on to the next portion of the array. This technique, discussed further in the next section, is called *blocking*, and 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 (e.g, dot-product). If the combined size of the arrays is smaller than the cache, we might expect only compulsory misses. However, this ideal case only occurs if the two arrays map to different cache sets (Figure 2b). If they overlap, either partially or entirely (Figure 2c), 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. As discussed in the next section, one solution is to change the memory allocation by merging the two arrays into an array of structures.

[1] Hill defines compulsory, capacity, and conflict misses in terms of miss ratios. When generalizing this concept to individual cache misses, we must introduce *anti-conflict* misses which miss in a fully-associative cache with LRU replacement but hit in an A-way set-associative cache. Anti-conflict 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.

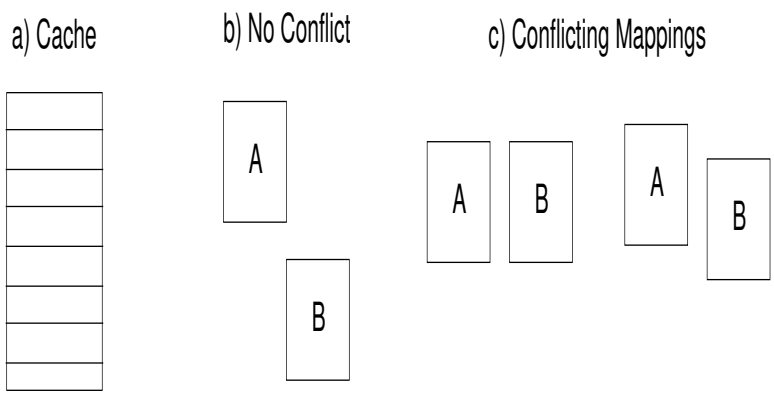


Figure 2: Conflicting Cache Mappings

The presence of conflict misses indicates a mapping problem. Figure 2b shows how two arrays that fit in cache with a mapping that will not produce any conflict misses, whereas Figure 2c shows two mappings that will result in conflict misses.

Our discussion thus far assumes a cache indexed using virtual addresses; many systems index their caches with real or physical addresses, making cache behavior strongly dependent on page placement. However, many operating systems use page coloring to minimize this effect, thus reducing the performance difference between virtual-indexed and real-indexed caches [6].

Techniques for Improving Cache Behavior

The analysis techniques described in the previous section can help a programmer understand the cause of cache misses. In this section, we present a *cookbook* of simple program transformations that can help eliminate some of the misses.

Program transformations can be classified by the type of cache misses they eliminate. Conflict misses can be reduced by array merging, padding and aligning structures, structure and array packing, and loop interchange. The first three techniques change the allocation of data structures, whereas loop interchange modifies the order that 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 [7, 10], structure and array packing, and loop interchange. In the following sections we present examples of each of these techniques, except loop interchange which was dis-

cussed in the introduction.

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 (Example 1). 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 (Example 2).

```
/* 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];
```

Example 1. Merging Arrays in C

```
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)
```

Example 2. Merging Arrays in FORTRAN 77

Padding and Aligning Structures

Referencing a data structure that spans two cache blocks may incur two misses, even if the structure itself is smaller than the block size. Padding structures to a multiple of the block size and aligning them on a block boundary can eliminate these ‘‘misalignment’’ misses, which generally show up as conflict misses. Padding is easily accomplished in C (Example 3) 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 (Example 4). Note that some dynamic memory allocators (e.g., some versions of *malloc()*) return cache-block aligned memory.

```
/* 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];
};
```

Example 3. Padding Structures in C

```
/* 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 + 15)/16)*16
```

Example 4. Aligning Structures in C

Packing

Packing is the opposite of padding; by packing an array into the smallest space possible the programmer increases spatial locality, which can reduce both conflict and capacity misses. In the example below (Example 5) 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 Example 6 permits 16 elements per block, rather than 4, reducing the maximum number of cache misses by a factor of 4.

```
/* 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;
```

Example 5. Unpacked Array in C

```
/* 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;
```

Example 6. Packed Array Structures in C

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 together, a programmer increases the program's temporal locality and frequently reduces the number of capacity misses. The example below combines two doubly-nested loops so that all operations are performed on an entire row before moving on to the next.

```
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];
```

Example 7: Separate Loops

```
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];
    }
```

Example 8: Fused Loops

Note that loop fusion is exactly the 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 paper 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 hence reduce capacity misses. The SPEC matrix multiply (part of `dnasa7`, a FORTRAN77 program) implements a column-blocked algorithm (Example 10) that achieves a 2.04 speedup over a naive implementation (Example 9) on a DECstation 5000/125. The algorithm tries to keep 4 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 4 columns do not fit in the cache, we could use a two-dimensional (row and column) blocked algorithm instead.

```
      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
```

Example 9. Naive Matrix Multiply

```
      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
```

Example 10. SPEC Column-Blocked Matrix Multiply

CPROF: A Cache Profiling System

The analysis and transformation techniques described above can help a programmer develop algorithms that minimize cache misses. However, cache misses result from the complex interaction between algorithm, memory allocation, and cache configuration; when the program is executed, the programmer's expectations may not match reality. We have developed a cache profiling system, CPROF, that addresses this problem by identifying where cache misses occur in a program and classifying them as compulsory, capacity, and conflict. This tool helps provide the insight necessary for programmers to select program transformations that improve cache behavior.

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

There are a number of cache and memory system profilers that differ in the level of detail they present to a programmer. High-level tools, such as MTOOL [2], identify procedures or basic blocks that incur large memory overheads. Other cache profilers, such as PFC-Sim [1] and CPROF, identify cache misses at the source line level, allowing much more detailed analysis. Of course this extra detail does not come for 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 both MemSpy [4] and CPROF. MemSpy [4] is very similar to CPROF, 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 only two miss types for uniprocessors: compulsory and replacement. Determining if a replacement miss is a result of referencing more data than will fit into the cache—a capacity miss—or a mapping problem—a conflict miss—is left to the user. MemSpy provides some insight into the cause of replacement misses by identifying the data structures competing for space in the cache. CPROF is unique in that it provides fine-grain source identification, data structure support, and classifies misses as compulsory, capacity, and conflict.

CPROF uses a flexible X-windows interface (see Figure 3) 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 sorted in descending order of importance, allowing quick identification of poor cache behavior. Misses are cross-referenced, so a programmer can quickly determine which of several data structures on a source line is

responsible for most cache misses.

CPROF annotates both static and dynamic data structures. Dynamically allocated structures are labeled by concatenating the procedure names on the call stack at the point of allocation [12]. 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 next lower number of cache misses. The *detail window* displays the number of each miss type for the currently selected source line or data structure.

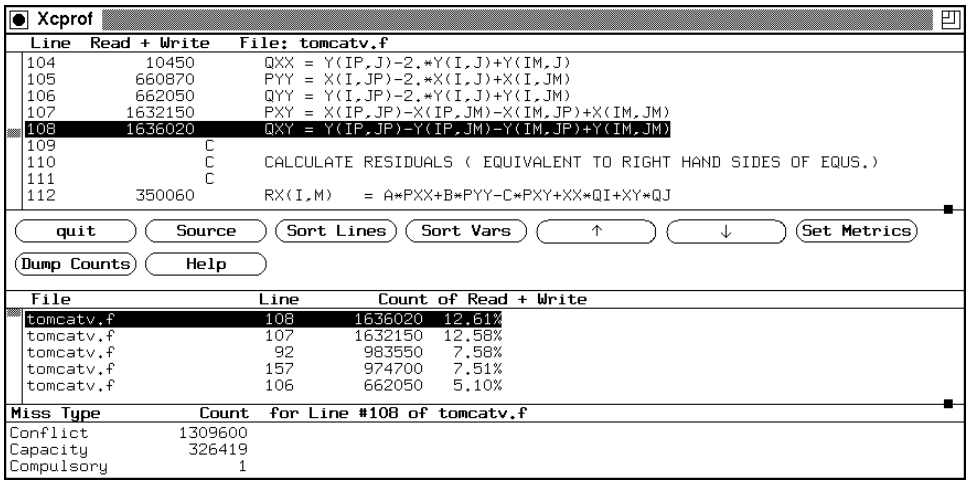
CPROF is very effective at identifying where a program exhibits poor cache behavior and the cache miss types help a programmer select what type of program transformation to apply. In the next section we describe more about how to use CPROF to tune program performance.

Case Study: The SPEC Benchmarks

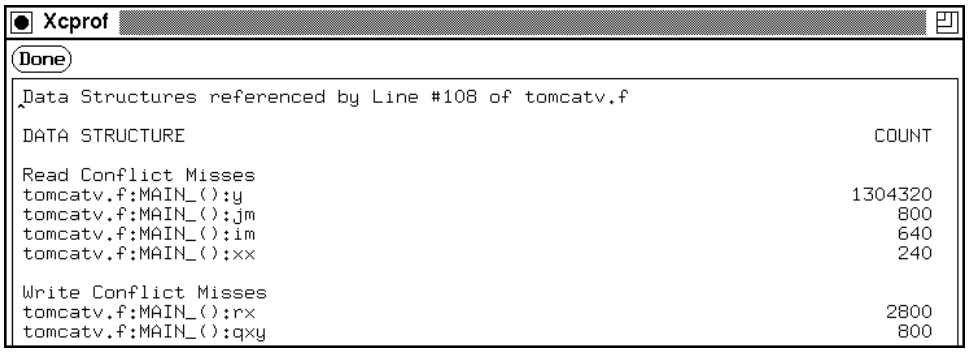
In this section, we describe a study where we used CPROF and our cookbook of transformations to tune the cache performance of six programs from the SPEC92 benchmark suite: `compress`, `dnasa7`, `eqntott`, `spice`, `tomcatv`, and `xlisp`. The purpose of this section is two-fold. First, we show that we 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[2] on three models of the DECstation 5000, the 5000/240, 5000/125, and 5000/200. Each of these machines have separate 64-kilobyte 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 these two machines is the cache miss penalty—16 processor cycles on the DECstation 5000/200 and 34 cycles on the DECstation 5000/125—which helps illustrate the importance of cache profiling as cache miss penalty increases. The 5000/240 uses a 40 MHZ MIPS

[2] System time accounts for very 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.



a.



b.

Figure 3: CPROF User Interface

CPROF's user interface (Figure 3a) 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-arrow 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 3b). 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.

R3000 processor chip and has a 28 cycle miss penalty.

These machines also have secondary differences with significant performance impact. For example, the 5000/2xx 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, which we compiled at optimization level -O2 per the SPEC make file. Note that, 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 (i.e., incorrect line numbers), since CPROF uses the same symbol table information.

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

Program	Restructuring Technique					
	Merging Arrays	**Loop Fusion**	**Loop Interchange**	**Padding & Aligning**	**Packing**	**Blocking**
btrix (dnasa7)	●	●	●			
cholesky (dnasa7)			●			
compress	●					●
eqntott					●	
gmtry (dnasa7)			●			
mxm (dnasa7)						●
spice	●					
tomcatv	●	●				
vpenta (dnasa7)	●	●	●			
xlisp				●		

Table 1. Program Restructuring Techniques

This table summarizes the restructuring techniques we use to improve the cache behavior of each program studied.

them separately.

Table 2 and Figure 4 present execution time results for the six benchmarks. The full programs execute as much as 90% faster when modified to improve cache behavior. Breaking out the kernels in `dnasa7` shows even

Program		Machine						Modification
		5000/125		5000/200		5000/240		
		Seconds	Speedup	Seconds	Speedup	Seconds	Speedup	
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 & 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 & 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 char
spice		2242.10		1762.34		1557.90		original
		1781.72	1.26	1406.04	1.25	1163.42	1.34	merged pointer & number
tomcatv		221.20		161.20		137.30		original
		167.24	1.32	134.38	1.20	91.40	1.50	merged arrays X & 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

Table 2: Execution Time Speedup

Both the original and tuned times for dnasa7 include the SPEC version of matrix multiply (mxm).

more striking results, with speedups as much as 3.46 for `vpenta` on the DECstation 5000/240, 2.53 on the DECstation 5000/125, and 2.14 on the DECstation 5000/200.

Below we discuss our experience cache profiling and modifying each program. We provide a very brief description of the program followed by the results of the initial CPROF execution. We then discuss the modifications we performed and the resulting speedups.

compress

`compress` is a UNIX utility that 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 quite large (69001 entries), to reduce the probability of collisions. When a collision does occur, a secondary probe is initiated.

CPROF indicates two source lines that reference the data structure that stores the keys are responsible for 71% of the cache misses. One source line is the initial probe into the hash table, which accounts for 21% of the cache misses. The other source line performs the secondary probe operation when there is a collision and accounts for 50% of the misses. CPROF also shows that most of the misses are capacity misses. Recall that 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 69001 to 5003, which is small enough to fit in the data cache. This change results in speedups of 1.92 on a DECstation 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 has 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% of the misses to the key array and 19% of the misses to the value array. The X-windows interface of CPROF allowed us to quickly notice that the array index is the same for both of these 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

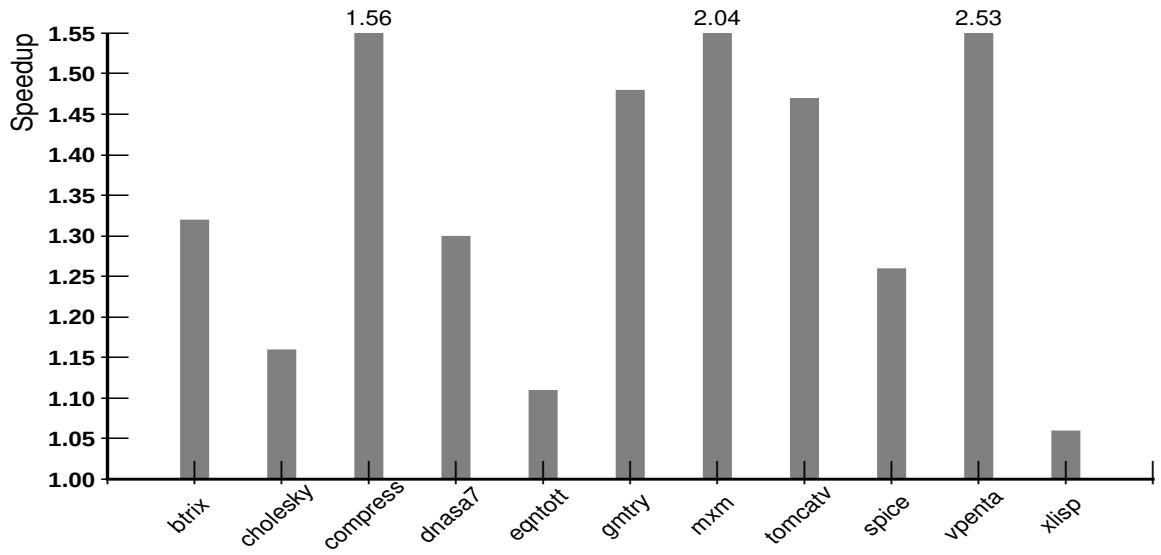


Figure 4: Speedups on a DECstation 5000/125 Obtained via Cache Profiling

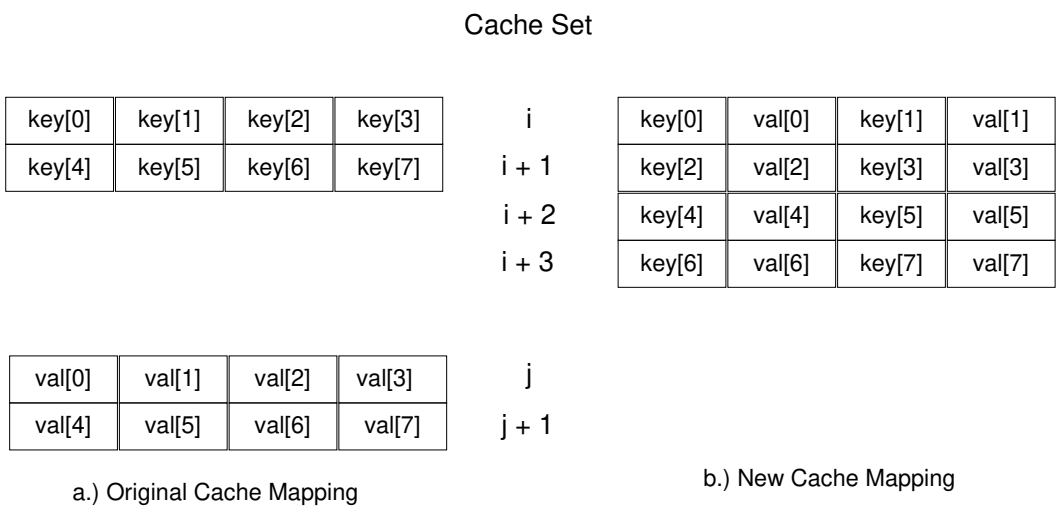


Figure 5: Cache Mappings for Compress

The initial allocation strategy for the key and value arrays (Figure 5a) resulted in as many as two cache misses for each successful hash table probe. Merging the two arrays into an array of structures (Figure 5b) effectively interleaves the elements of the two arrays and results in only one cache miss per successful probe.

likely to reference the corresponding value, which resides in the other array and hence a different cache block (see Figure 5a).

Merging the two arrays into a single array of structures places the key and value in the same cache block (see Figure 5b), 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 DECstation 5000/240, 1.05 on the 5000/125 and 1.02 on the 5000/200.

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% of its time in the quick-sort routine [9]. CPROF further reveals that most of this time is spent moving the sort keys from memory into the cache; over 90% 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 #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. However, the trade-off between fewer cache misses and the time to unpack the data, is implementation dependent.

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% of the cache misses occur during the mark and sweep garbage collection, most of which are conflict misses. During this phase, the program first traverses the reachable nodes and marks them accessible, 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. However, such an extensive modification was outside the scope of this study.

CPROF shows that 19% 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. By placing 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, we see that the node structures allocated on line #540 of `xldmem.c` incur a large number of conflict misses. Inspection of the source reveals each node structure occupies 12 bytes, or three-fourths of a 16-byte cache block. Consequently, only half of the nodes reside entirely within a single cache block (see Figure 6). The remaining half of the nodes reside in two contiguous cache blocks, potentially causing two cache misses—when referenced—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 DECstation 5000/240, 1.06 on the 5000/125 and 1.03 on the 5000/200.

It's important to realize that 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 (if not in cache), rather than only half. Similarly, while many memory allocators (e.g., the ULTRIX `malloc()` routine) return cache-block-aligned memory, `xlisp` pre-allocates 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.

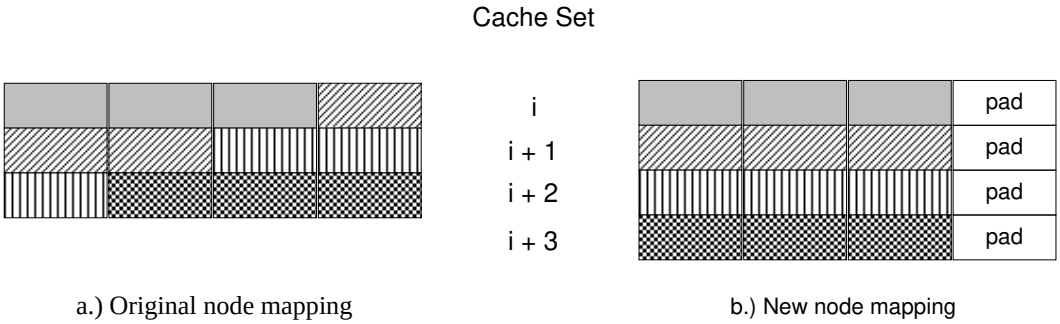


Figure 6: Cache Mappings for Xlisp Node Structures

Each pattern corresponds to a different node structure, while *pad* indicates wasted storage. The initial allocation strategy (Figure 6a) 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 (Figure 6b) reduces this to only one cache miss per node not resident in the cache.

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% increase in storage. This increase could adversely affect virtual memory performance for larger programs, although this was not a problem with this input [9].

tomcatv

`tomcatv` is a FORTRAN 77 mesh generation program that uses seven two-dimensional data arrays, each of which requires approximately 0.5 M-Byte. The algorithm (see Figure 7) 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 in these errors.

Since the arrays are much larger than the cache and the arrays 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. It is easily observed from the source code that the two arrays are always referenced with the same indices. Hence, to improve spatial locality, we merged them together, placing elements X(I,J) and Y(I,J) in the same cache block. This modification results in speedups of 1.50 on the DECstation 5000/240, 1.32 on the 5000/125 and 1.20 on the 5000/200.

Running CPROF on the modified `tomcatv` finds that capacity misses to the RX and RY arrays now dominate. As Figure 7 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 which 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 additional 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, recall that each array is 0.5M-Bytes in size, which is much larger than the 64K-Byte data cache. Hence, the elements referenced at the start of one loop are not in the cache at the start of the next loop.

The solution is to improve temporal locality by restructuring the program so that all allowable operations are performed on an element when it is resident in the cache. Transforming the program via loop fusion (see Figure 7) merges these loops so the program contains only one forward loop and one backward loop. We can not perform the

```
     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
```

Original Tomcatv

```
     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
     endfor
```

Loop Fused Tomcatv

Figure 7: Tomcatv Psuedo-Code

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 operations of the forward wave on one row of the arrays. This results in speedups of 1.60, 1.47, 1.28 on the DECstation 5000/240, 5000/125, and 5000/200 respectively.

operations of both the forward pass and backward pass in the same loop because of data dependencies. Notice that 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 DECstation 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 row number arrays cause 34% of the cache misses. Another two source lines accessing the column pointer and column number arrays contribute an additional 12% of the cache misses. Each pair of source lines is contained in a small loop that locates an element (I,J) in the sparse matrix. CPROF shows that the majority of the misses caused by these source lines are conflict misses, indicating a mapping problem. Again, the X-windows interface of CPROF allows us to quickly observe 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 DECstation 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, 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 very low miss ratio on a 64-Kbyte data cache (0.8%) and shuffling FFTs have inherently poor cache performance. The speedup we obtained for the entire collection of kernels is 1.51 on the DECstation 5000/240, 1.30 on the 5000/125, and 1.24 on the 5000/200.

vpenta

The `vpenta` kernel simultaneously inverts 3 pentadiagonals, a routine commonly used to solve systems of partial differential equations. CPROF first finds that the miss ratio is a startling 36%, mostly due to conflict misses. Using CPROF to identify the mapping problems, we discovered two nested loops responsible for over 90% of the cache misses. One loop accesses three arrays while the other accesses 8 arrays. Recall that we can eliminate conflict misses by changing the allocation of data structures or the order that they are accessed. Inspection of the source code reveals that both of these techniques can be applied. We discovered the loops could be interchanged to traverse the arrays in column order and also identified three opportunities for array merging. These modifications result in speedups of 2.93 on a DECstation 5000/240, 2.10 on a 5000/125 and 1.85 on the 5000/200. It is interesting to note that the original code runs slower on the 5000/240 than on the 5000/200, despite the 60% 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 100ns 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 over 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 to improve temporal locality by eliminating multiple passes over the same arrays 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 a large number of capacity misses in two nested loops. Inspection of the source code 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 DECstation 5000/240, 1.16 on the 5000/125 and 1.13 on the 5000/200. Blocking can also be applied to cholesky [7], but we chose to apply a much simpler transformation.

btrix

`btrix` is a tri-diagonal solver. CPROF shows that most of the 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. Notice that we were able to apply several transformations after a single run of CPROF. On the DECstation 5000/240 we obtain a speedup of 1.48, 1.32 on the 5000/125 and 1.27 on the 5000/200.

gmtry

`gmtry` is a kernel dominated by a Gaussian elimination routine. CPROF finds 99% 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 DECstation 5000/240, and 1.48 on both 5000/200 and 5000/125.

```
      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
```

Original Gaussian Elimination

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

Interchanged Gaussian Elimination

mxm

`mxm` is a matrix-matrix multiply routine. The naive matrix multiply algorithm is a well-known ‘‘cache buster’’, because there is little data re-use between loop iterations. The SPEC `mxm` implementation does not use this simple algorithm, instead using a column-blocked implementation (described in the "cookbook" earlier) that re-uses the same four columns throughout the two inner-most loops. It is interesting to note that improving cache performance was not the original rationale for blocking `mxm`; instead, the intent was to improve the opportunity for vectorizing compilers to 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 over the naive algorithm on a DECstation 5000/240, 2.04 on the 5000/125 and 1.74 on a DECstation 5000/200. For larger matrices, a two-dimensional (row and column) blocked algorithm would perform better, but for the standard SPEC input size the extra overhead decreases performance.

Summary

We have demonstrated how cache profiling and program transformation can be used to obtain significant speedups on six of the SPEC92 benchmarks. The speedups range from 1.02 to 3.46, depending on the machine’s memory system, with greater speedups obtained in the FORTRAN programs. Since FORTRAN77 does not support structures many of the programs exhibit poor spatial locality. Improving the spatial locality by interleaving the elements of disjoint arrays provided substantial improvements in most of the FORTRAN programs. Using FORTRAN90, which provides structures, would greatly simplify these transformations. CPROF was very effective at identifying when to merge arrays. Loop interchange also improved the spatial locality in the FORTRAN programs; in many programs, loop interchange is a trivial transformation that can be easily identified by inspection. The temporal locality of FORTRAN programs was improved by loop fusion, which requires programmers to perform all

allowable operations on data while in the cache versus performing each operation in turn on all the data. It is important to remember that some of the transformations discussed in this paper may be counter-productive on machines with vector registers.

The C programs benefited from padding and alignment of structures, merging arrays into an array of structures, and changing the declaration of a variable to pack more elements into a single cache block. Notice, that padding and packing are opposite approaches and which to use is dependent on the program being profiled.

Conclusion

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. Unfortunately, actual cache performance does not always match the programmer's expectations, and many programs are too complex to fully analyze the interactions between memory reference patterns, data allocation, and cache organization. In these cases, a tool like CPROF becomes an important element in a programmer's tool box. CPROF provides cache performance information at the source line and data structure level allowing a programmer to identify hot-spots. The insight CPROF provides, by classifying cache misses as compulsory, capacity, and conflict, helps programmers select appropriate program transformations that improve a program's spatial or temporal locality, and thus overall performance.

Acknowledgements

We would like to thank Karen Miller, Alain Kagi, and Chris Maguire for work on an earlier version of CPROF. Alain Kagi and Scott Kempf found several bugs in the current 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, James Larus, and Mark Hill provided helpful comments and suggestions on early drafts of this paper. 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-9225097, donations from A.T.&T. Bell Laboratories, Digital Equipment Corporation, and Xerox Corporation, and the Graduate School of the University of Wisconsin.

References

1. D. Callahan, K. Kennedy, and A. Porterfield, "Analyzing and Visualizing Performance of Memory Hierarchies," *Instrumentation for Visualization*, ACM Press, (1990).
2. A. J. Goldberg and J. Hennessy, "Performance Debugging Shared Memory Multiprocessor Programs with MTOOL," *Proceedings Supercomputing '91*, pp. 481-490 (November 1991).
3. S. L. Graham, P. B. Kessler, and M. K. McKusick, "An Execution Profiler for Modular Programs," *Software Practice & Experience* **13** pp. 671-685 (1983).
4. A. Gupta, M. Martonosi, and T. Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," *Performance Evaluation Review* **20**(1) pp. 1-12 (June 1992).
5. M. D. Hill and A. J. Smith , "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers* **38**(12) pp. 1612-1630 (December 1989).
6. R. E. Kessler and Mark D. Hill, "Page Placement Algorithms for Large Real-Index Caches," *ACM Trans. on Computer Systems* **10**(4) pp. 338-359 (November 1992).
7. M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proceedings ASPLOS IV*, pp. 63-74 (April 1991).
8. SPEC Newsletter, , Standard Performance Evaluation Corporation, Fairfax, VA (December 1991). SPEC c/o NCGA, Suite 200, 2722 Merilee Drive, Fairfax, VA 22031
9. D. N. Pnevmatikatos and M. D. Hill, "Cache Performance of the Integer SPEC Benchmarks on a RISC," *ACM SIGARCH Computer Architecture News* **18**(2) pp. 53-68 (June 1990).
10. A. Porterfield, "Software Methods for Improvement of Cache Performance on Supercomputer Applications," Ph. D. Thesis, Dept of Computer Science, Rice University, (1989).
11. A. J. Smith, "Cache Memories," *ACM Computing Surveys* **14**(3) pp. 473-530 (Sept. 1982).
12. B. Zorn and P. N. Hilfinger, "A Memory Allocation Profiler for C and Lisp," *Proceedings of the Summer 1988 USENIX Conference*, (June 1988).