SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(2), 167–172 (FEBRUARY 1997)

Case Studies on Cache Performance and Optimization of Programs with Unit Strides

pei-chi wu and kuo-chan huang
Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta-Hsueh Road, Hsinchu, Taiwan, Republic of China
(email: {pcwu, kchuang}@csie.nctu.edu.tw)

SUMMARY

Cache performance in modern computers is important for program efficiency. A cache is *thrashing* if a significant amount of time is spent moving data between the memory and the cache. This paper presents two cache thrashing examples, one in scientific computing and one in image processing, both of which involve several one-dimensional arrays that are accessed sequentially, i.e., with unit strides. Accessing arrays in unit strides was considered very efficient on cache-based computer systems. However, the existence of cache thrashing is demonstrated by significant increases in computing speed in the equivalent programs tuned for cache locality. This shows that accessing several arrays sequentially may cause cache thrashing. Thus, to improve cache performance, it is important that the compiler or the programmer takes all arrays inside a loop into consideration. © 1997 by John Wiley & Sons, Ltd.

key words: computer architecture; cache performance; code optimization; cache thrashing

INTRODUCTION

Cache performance in modern computers is important for program efficiency. Many research and development projects have been devoted to cache hardware design.[1] There are many trade-offs (on cache design), e.g., capacity and efficiency. On the other hand, program efficiency is also dependent on compiler and programming techniques. An example is the development of *block algorithms*[2] for Basic Linear Algebra Subprograms (BLAS)[3]: a speedup of 4·3 has been reported[4] by applying block algorithms to matrix multiplication. This result indicates that most of the computing time has been spent handling cache misses in the original matrix multiplication algorithm.

A cache is described as *thrashing* if a significant percentage of the computing time is spent moving data between the memory and the cache. In most current computing environments, cache miss rates are not available in program profiling. Thus, cache thrashing may occur very often, consume much computing time, and yet remain undetected. Gannon *et al*.[5] described a method to estimate cache performance in programs with nested loops of array accesses. This method assumed that the cache is entirely under the control of the compiler. Because the assumption does not stand in most computer systems, the method is not easy to apply. Brewer *et*

CCC 0038–0644/97/020167–06

Received 6 March 1995
Revised 28 May 1996

```
#define ARRAY_SIZE (1<<17)
float a[ARRAY_SIZE], b[ARRAY_SIZE], c[ARRAY_SIZE], d[ARRAY_SIZE];
float e[ARRAY_SIZE], f[ARRAY_SIZE], g[ARRAY_SIZE], h[ARRAY_SIZE];
float e1[ARRAY_SIZE], f1[ARRAY_SIZE], g1[ARRAY_SIZE], h1[ARRAY_SIZE];
float e2[ARRAY_SIZE], f2[ARRAY_SIZE], g2[ARRAY_SIZE], h2[ARRAY_SIZE];

/* main loop */
for(i=0; i<ARRAY_SIZE; i++)
{
  a[i] = b[i] + c[i] + d[i];
  e[i] = f[i] - g[i] - h[i];
  e1[i] = f1[i] - g1[i] - h1[i];
  e2[i] = f2[i] - g2[i] - h2[i];
}
```

Figure 1. Program **sci-a**

al.[6] developed the MAPI tool for the analysis of memory access patterns, especially for matrix algorithms. Callahan *et al.*[7] developed the PFC-Sim tool for measuring the cache performance of a set of computational-intensive Fortran programs, and applied several loop transformation techniques to these programs. Lam *et al.*[4] evaluated several optimizations to improve cache performance of blocked matrix multiplication algorithms. Gannon and Jalby[8] presented an analytical model of hierarchical memory system and performance results of Fast Fourier Transform algorithms. Bailey[9] discussed unfavorable strides in Fortran programs that access multi-dimensional arrays.

Most prior research work has focused on the cache performance of programs with non-unit strides of multi-dimensional arrays. For example, the strides in matrix multiplication are usually in the size of a row or a column, and the strides in Fast Fourier Transform are usually in powers of two. This paper presents two cache thrashing examples, one in scientific computing and one in image processing. Both programs access several large one-dimensional arrays sequentially, i.e., with unit strides. Accessing arrays in unit strides was considered very efficient on cache-based

```
#define ARRAY_SIZE (1<<17)

struct {
  float a, b, c, d;
  float e, f, g, h;
  float e1, f1, g1, h1;
  float e2, f2, g2, h2;
} A[ARRAY_SIZE];

/* main loop */
for(i=0; i<ARRAY_SIZE; i++)
{
  A[i].a = A[i].b + A[i].c + A[i].d;
  A[i].e = A[i].f - A[i].g - A[i].h;
  A[i].e1 = A[i].f1 - A[i].g1 - A[i].h1;
  A[i].e2 = A[i].f2 - A[i].g2 - A[i].h2;
}
```

Figure 2. Program **sci-b**

computer systems. However, the existence of cache thrashing is demonstrated by significant increases (e.g., the speedup > 1) in computing speed in the equivalent programs tuned for cache locality. This shows that accessing several arrays sequentially may cause cache thrashing. Thus, to improve cache performance, it is important that the compiler or programmer takes all arrays inside a loop into consideration when analyzing memory access patterns in a program.

TWO EXAMPLE PROGRAMS

This section presents two example programs. The codes presented are the core parts of real application programs from specific application domains, including particle simulation and image processing. The cache performance of both kinds of program has not been considered carefully before.

A particle simulation program usually simulates many particles, e.g., 10^5 particles. Each particle contains several properties, such as its three-dimension velocity. Because such a scientific simulation program is usually coded in Fortran, and Fortran 77 does not support data abstraction (e.g., the 'structure' data types in the C language), the programmer can only use several large arrays, each of which represents one property of these particles. The program then accesses these arrays to update each particle during the simulation. Figure 1 (program **`sci-a`**) shows the core of a particle simulation program. Instead of using Fortran, we use the C language here.

Because the **`ARRAY_SIZE`** in Figure 1 is set to be a power of 2 (2^{17}), these arrays (**`a, b, c, . . .`**) are likely have the same patterns in their low-order address bits. Set-associative caches use middle-order address bits as a set index. These arrays may then be mapped to the same set, and accessing **`a[i], b[i], c[i]`**, . . ., may cause *conflict misses*. Because the **`sci-a`** program accesses 16 arrays, it may also cause cache thrashing on the system with a set-associative degree less than 16. This problem cannot be completely solved by setting the size to 10^5, for example, because how these arrays are arranged in the memory is dependent on the linker or compiler. Another example is that dynamic memory allocators (e.g., Haertel's allocator, also

Table I. Performance results on four workstations. (σ = **`standard deviation`**)

Program	Sparc-2	HP 9K/720	IBM RS6K/590	DEC Alpha3K/500
sci-a (mean)	156.0s	107.6s	77.7s	17.7s
sci-a (σ)	1.21	0.83	1.89	0.05
sci-b (mean)	69.2s	25.5s	2.8s	12.2s
sci-b (σ)	0.69	0.13	0.05	0.02
Speedup	2.3	4.2	27.8	1.5
im-a (mean)	430.6s	310.6s	19.4s	50.2s
im-a (σ)	3.35	0.25	0.25	0.20
im-b (mean)	91.8s	54.7s	19.3s	19.0s
im-b (σ)	0.99	0.09	0.23	0.14
Speedup	4.7	5.7	1.0	2.6

```
#define IMAGE SIZE (512*512)
char A[IMAGE SIZE];
char IP[IMAGE SIZE];

/* loop for 2 images */
for(i=0; i<IMAGE SIZE; i++)
  IP[i] = IP[i]  | A[i];
```

Figure 3. Program **im-a**

called GNU Local and briefly described in Grunwald *et al.*[10]) may arrange any requested large chunks (e.g., chunk size > 4096 bytes) starting at addresses of powers of two.

Figure 2 shows the **sci-b** program, an equivalent program of **sci-a**. Program **sci-b** uses a structure containing fields **a, b, c,** . . ., and an array **A[ARRAY_SIZE]** to store all the particles. Accesses to **A[i].a,A[i].b,A[i].c**, . . . are adjacent and can be handled easily in the caches of most computer systems.

Figure 3 shows an image processing program (**im-a**) where two images are combined by the bitwise-or operation (operator '|' in the C language). There is a loop for writting the resulting image to one of the input images (**IP** = **IP** | **A**). Because the **IMAGE_SIZE** is 512*512, a power of 2, accesses to **A[i]** and **IP[i]** may also cause cache misses. Figure 4 shows an equivalent program (**im-b**) that adjusts the size of arrays **A** and **IP**. This adjustment assumes that arrays **A** and **IP** are placed adjacently. Note that this adjustment is not guaranteed to work well in all computer systems. Many users of supercomputers have applied this (or a similar) technique: the first dimension of multi-dimensional arrays is declared to be slightly larger than a power of two[9].

PERFORMANCE RESULTS

Table I shows the performance results on four workstations, including a Sun SPARC-2 station, a HP 9000/720 workstation, an IBM RS6000/590 and a DEC Alpha3000/500. The C compilers used are provided by the vendors, except that the GNU C compiler is used on the HP 9K/720. Each floating-point number in **sci-a** and **sci-b** occupies four bytes on all four machines. All program codes and data are loaded in memory, and there are few page faults. The computing time (shown in seconds) is obtained by the **clock()** function and the **time** shell command. All the programs run on lightly loaded environments. The loops in **sci-a** and **sci-b** are iterated 100 times, and the loops in **im-a** and **im-b** are iterated 1000 times. Each of the four programs is executed 30 times to calculate the average execution time and the standard deviation. In Table I, seven of the eight 'speedup' entries are

```
#define IMAGE SIZE (512*512)
#define ARRAY_SIZE (IMAGE SIZE+13*4)
char A[ARRAY_SIZE];
char IP[ARRAY_SIZE];

/* The loops remain unchanged. */
```

Figure 4. Program **im-b**

Table II. Cache architectures of four workstations

Cache parameters	Sparc-2	HP 9K/720	RS6K/590	DEC Aplpa 3K/500
cache size (byte)	64 K	256 K	256 K	8 K
block size (byte)	16	32	256	32
cache organization	direct mapping	direct mapping	4-way set associative	direct mapping

Table III.. Estimated cache misses of the four programs on the four machines

Program	Sparc-2		HP 9K/720		RS6K/590		DEC Alpha3K/500	
	Read	Write	Read	Write	Read	Write	Read	Write
sci-a	12×2^{17}	4×2^{17}	12×2^{17}	4×2^{17}	12×2^{17}	4×2^{17}	12×2^{17}	4×2^{17}
sci-b	4×2^{17}	0	2×2^{17}	0	2^{15}	0	2×2^{17}	0
sci-a/sci-b	4		8		64		8	
im-a	2×2^{18}	0	2×2^{18}	2^{18}	2^{11}	0	2×2^{18}	0
im-b	2^{15}	0	2^{14}	0	2^{11}	0	2^{14}	0
im-a/im-b	16		48		1		32	

greater than 1, and the maximal speedup is 27·8. This result indicates that the example programs have caused cache thrashing in many of the systems tested.

Table II shows the cache parameters of the four machines used in our experiment. The cache size counts only the data cache for machines with separate data and instruction caches. All machines except for RS6K/590 use direct mapping caches. RS6K/590 uses a four-way associative cache. DEC Alpha 3K/500 has a two-level cache. Here we list only its first level data cache.

Using Table II, we can estimate the number of cache misses that arose when the four programs ran on the four machines. Since the penalities of read and write misses may be different, we calculate the number of these two cache misses separately. Any array access in the **`sci-a`** program causes one cache miss in the four machines, so in total there are 12×2^{17} read misses and 4×2^{17} write misses, respectively. The **`sci-b`** program accesses array **`A`** sequentially, so larger cache blocks result in fewer cache misses. The program **`im-a`** accesses two arrays (**`A`** and **`IP`**) of the same low-order address bits, so any machine with a direct mapping cache may cause cache misses when accessing these arrays. The number of cache misses depends on how compilers arrange the loading sequence of **`IP[i]`** and **`A[i]`**. On HP9K/720, the GNU C compiler generates code that first loads **`IP[i]`** and then **`A[i]`** to registers, and finally stores the result in **`IP[i]`**. There are two read misses and one write miss in each iteration: 2×2^{18} read misses and 2^{18} write miss in total. On the other hand, the C compilers on Sparc-2 and DEC Alpha generate codes that first load **`A[i]`** and then **`IP[i]`** so there are totally 2×2^{18} read misses and no write misses. In program **`im-b`**, there are 3×2^{18} array accesses. Because accessing

IP[i] the second time is always a cache hit, the number of cache misses is: 2×2^{18}/block size. RS6K/590's four-way associative cache makes **im-a** as fast as **im-b**, since both programs access only two distinct arrays. However, when the number of arrays grows beyond 4, as 16 in **sci-a**, RS6K/590 generates the same number of cache misses as other machines. The estimated numbers of cache misses are summarized in Table III.

Table III also lists the ratios of total cache misses: rows **sci-a**/**sci-b and im-a/im-b**. These ratios explain the speedups obtained in Table I. The largest cache miss ratio results in **sci-a**/**sci-b** with RS6K/590, which has the best speedup 28·2 from **sci-a** to **sci-b**. DEC Alpha3K/500 has the same cache ratios as HP 9K/720, but its speedups in Table I are less than HP. This is because DEC Alpha3K/500 has a two-level cache and its cache miss penalty, the first level, is less than that of HP 9K/720.

CONCLUSION

In this paper, we have presented two case studies on cache performance of programs with unit strides. The cache thrasing of example programs is detected when the equivalent programs tuned for cache locality achieve a significant speedup. This result shows that accessing several arrays sequentially may also cause cache thrashing. Thus, to improve cache performance, it is important that the compiler or programmer takes all arrays inside a loop into consideration.

REFERENCES

1. J. L. Hennessy and D. A. Patterson, *Computer Architecture—A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.
2. T. L. Freeman and C. Phillips, *Parallel Numerical Algorithms*, Prentice-Hall, Hemel Hempstead, 1992.
3. J. J. Dongarra, J. D. Croz, S. Hammarling and I. Duff, 'A set of level 3 basic linear algebra subprograms', *ACM Trans. Mathematical Software*, **16**(1) 1–17 (March 1990).
4. M. S. Lam, E. E. Rothberg and M. E. Wolf, 'The cache performance and optimizations of blocked algorithms', *Proc. 4th Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, 1991, pp. 63–74.
5. D. Gannon, W. Jalby and K. Gallivan, 'Strategies for cache and local memory management by global program transformation', *Journal of Parallel and Distributed Computing*, **5** 587–616 (1988).
6. O. Brewer, J Dongarra and D. Sorensen, 'Tools to aid in the analysis of memory access patterns for Fortran programs', *Parallel Computing*, **9**(1) 25–35 (December 1988).
7. D. Callahan, K. Kennedy and A. Porterfield, 'Analyzing and visualizing performance of memory hierarchies', in M. Simmons and R. Koskela (eds), *Performance Instrumentation and Visualization*, ACM Press, 1990.
8. D. Gannon and W. Jalby, 'The influence of memory hierarchy on algorithm optimization: programming FFTs on a vector multiprocessor', in *The Characteristics of Parallel Algorithms*, MIT Press, 1987.
9. D. H. Bailey, 'Unfavorable strides in cache memory systems', *RNR Technical Report RNR-92-015*, Numerical Aerodynamic Simulation System Division, NASA Ames Research Center, USA, 1992.
10. D. Grunwald, B. Zorn and R. Handerson, 'Improving the cache locality of memory allocation', *Proc. ACM SIGPLAN'93 Programming Language Design and Implementation* (also as *ACM SIGPLAN Notices*, **28**(6) 177–186 (June 1993)).