

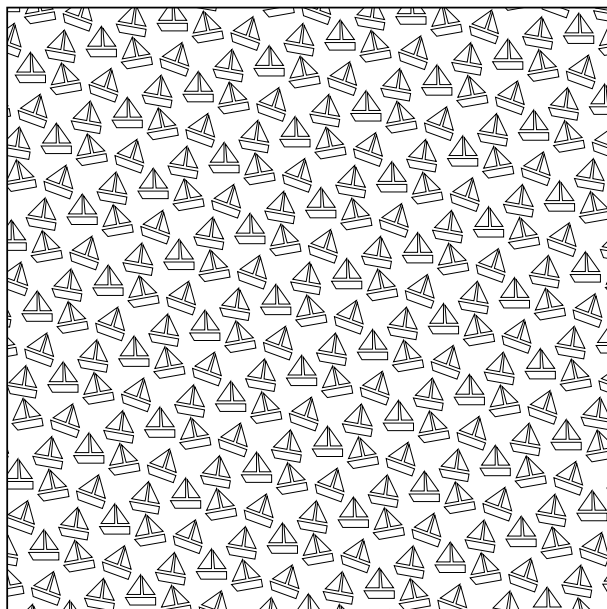
MACQUARIE COMPUTING REPORTS

Efficient Image Processing on RISC Workstations

Leonard G. C. Hamey

len@mpce.mq.edu.au

December 1992



Abstract

The use of Reduced Instruction Set Computers for scientific applications is common today. Advances in processor design and compiler technology make it possible to perform large-scale computations on RISC workstations. The RISC design provides simple instructions that operate at high speed and compiler optimisations employ the machine's capabilities to perform naively programmed operations with reasonable efficiency. However, there remain opportunities for improvement in execution performance by optimising the design of the high-level language program, and, conversely, there are also traps for the unwary programmer. Partial unrolling of loops and employing scalar variables to assist the compiler in the use of registers are two simple techniques that can yield performance improvements of up to 50% depending upon the compiler and the architecture. Unexpected performance penalties as high as 500% may be paid for cache conflicts, where two large data arrays are contending for the same cache locations. This report discusses how to improve the performance of critical code sections in scientific programs, and how to detect and avoid cache conflict performance penalties.

Contents

1	Introduction	2
2	Characteristics of RISC Architectures	3
3	Performance Improvement Techniques	3
3.1	Loop Unrolling	4
3.2	Using Scalar Variables	5
3.3	Combining the Techniques	8
4	Consider the Cache	11
4.1	Measuring Cache Parameters	13
4.2	Cache Performance Impact	14
5	Solving Cache Conflicts	15
5.1	Importance of the Memory Allocator	18
5.2	Randomisation of Location	19
5.3	Iiffe Vector Randomisation	20
5.4	Optimisation of Location	25
5.5	Bulk Data Transfers	26
5.6	A Hardware Solution	27
6	Prognosis	35
7	Conclusion	36
8	Acknowledgements	37
A	Program Listing	38

1 Introduction

Reduced Instruction Set Computers (RISC) are now in widespread use for all types of computational tasks. In the area of scientific computing, RISC workstations are being increasingly used for compute-intensive tasks such as digital signal and image processing. Advances in compiler design have made it possible for programmers to write high-level language code naively and yet obtain efficient execution—the compiler optimises the machine code to suit the underlying architecture. However, there remain opportunities for programmers to improve the execution performance by improving the design of the high-level language programs, and, conversely, there are also traps waiting to rob unwary programmers of the performance that they expect.

This report discusses some issues that are involved in obtaining peak efficiency in scientific computation on RISC architectures. Our attention is limited to highly regular computations, exemplified by the dot-product of two vectors and a one-dimensional convolution. The techniques discussed are applicable, however, to a wide range of regular computations such as digital signal and image processing and matrix and vector computations.

Section 3 discusses the well-known technique of loop unrolling, which can yield performance benefits of up to 35% on the examples discussed. Loop unrolling may be profitably combined with the use of scalar variables to improve the use of the RISC architecture's register set. This technique is also described in section 3.

One of the traps waiting for the unwary programmer is cache conflicts, which occur when the cache is unable to usefully retain the data needed by the program. This problem is most severe when two or more large arrays of data are being processed, as it is possible for cache conflicts to cause all memory accesses to be referred to main memory. Performance loss resulting from slower memory access time can be as bad as a factor of five. Using the example of dot-product computation, section 4 discusses the problem of cache conflicts, and ways to detect and avoid them.

The Sun SPARCstation performance results reported in this paper were obtained on programs compiled with the Sun C compiler version 1.1, which is part of the Sun Compiler 1.0 release. Compilation was performed with the `-fast -O3` optimisation options, and the `-fsingle` option to use single-precision computation whenever possible. The performance results for Digital workstations were obtained using the default Ultrix C compiler with the `-O` optimisation option and the `-float` option for single-precision computation. All floating-point performance measurements reflect actual single-precision computational speed, including looping overhead. The computations were performed using vectors of 2^{17} floating-point elements.

```

int factorial (n)
int n;
{
    int fact;

    fact = 1;
    for ( ; n > 1; n--)
        fact *= n;
    return fact;
}

```

Figure 1: A computation that will use registers only.

2 Characteristics of RISC Architectures

RISC design principles are well understood and discussed in many textbooks (e.g. [1, 4]). Central to the design of RISC architectures is the choice of a fast, small instruction set. Instructions that have traditionally been slow and cumbersome, such as procedure calls, are designed to operate faster because of their great importance in actual programs. Other operations that are of lesser significance in actual programs are not implemented in hardware at all. One common omission is integer multiplication and division. In the past, integer operations have often been substituted for floating-point computations to speed up tasks such as image processing; such techniques are inappropriate with today's RISC processors.

RISC architectures typically have a generous supply of registers. The SPARC architecture, for example, has 31 general-purpose 32-bit registers and 32 single-precision floating-point registers. These registers are used, however, as the source and destination for every arithmetic and logical operation—the only method of access to memory is via load and store instructions which transfer data between the registers and memory. The simplification of memory access is considered a benefit in RISC design because it reduces the complexity of the CPU and encourages the use of registers for scalar variables [4]. To obtain the efficiency of the RISC approach, compilers employ registers to hold scalar variables as much as possible. Thus, a program segment such as figure 1 needs no memory to store the variables `n` and `fact` but rather retains them in registers throughout the computation.

3 Performance Improvement Techniques

There are many ways to modify high-level language programs so as to improve the execution performance of the compiled program. Which techniques are

effective depends strongly upon the compiler and the underlying machine architecture. It is often useful to employ a profiling tool (such as `gprof`) to determine which portions of a program are consuming the most time. The critical code sections should then be improved.

The first option to consider is a more efficient algorithm that performs the required task. However, if a better algorithm is not available, it may still be possible to improve the performance of the existing algorithm. In some cases, careful consideration of the computation being performed may yield performance improvement. For example, conversions between integer and floating-point are time consuming—unless overflow is a potential problem, the mean of an array of integers may be computed much more rapidly by computing an integer sum than by computing a floating-point sum. In other cases, it may prove useful to examine the machine (assembly) code produced by the compiler to determine whether the program is being compiled efficiently—this may help identify specific problem areas that can be overcome in the high-level language code.

There are also some approaches have widespread usefulness. One such approach is loop unrolling, which has long been known as an effective means of gaining improved performance. Another approach which is particularly suited to RISC architectures is to assist the compiler in using registers by loading data from arrays and structured variables into scalar variables.

3.1 Loop Unrolling

Loop unrolling is a simple heuristic for improving the performance of tight computational loops such as a dot-product (figure 2). The body of the loop is unrolled several times (e.g. eight times) yielding a new loop that performs eight iterations of the old loop in each iteration. By performing counter and pointer increments and the loop test only once in the unrolled loop, the looping overhead is reduced relative to the computation achieved and performance is enhanced. Loop unrolling is only applicable to small loops that are executed many times, and it may have a detrimental effect on some architectures.¹ Ultimately, performance benefits can best be determined experimentally.

Figure 3 contains an unrolled version of the dot-product computation. Note that the pointer increments are performed only once at the end of the unrolled loop while within the loop, array subscripts are used to access successive elements of the two vectors. In RISC architectures such as the SPARC (used in the Sun architectures investigated in this report) and MIPS (used in the Digital architectures investigated in this report), small constant pointer offsets are efficiently handled and incur no speed penalty when compared with direct pointer dereferencing. However, pointer increments require a separate instruction, so it is preferable to avoid performing them unnecessarily.

¹Loop unrolling may be detrimental on architectures which have a small instruction cache, since the unrolled loop may exceed the size of the cache. Loop unrolling may also be detrimental on architectures which employ special tricks to speed up tight loops.

```

float dot_normal (v1, v2, n)
float *v1, *v2;
int n;
{
    float result = 0.0;
    int i;

    for (i = n; --i >= 0; )
        result += *v1++ * *v2++;
    return result;
}

```

Figure 2: A straight-forward dot-product implementation.

Table 1: Measured dot-product performance in Mflop.

Architecture	Normal	Unrolled	Improvement
Sun 1+/IPC	2.4	3.1	29%
Sun ELC	3.5	4.7	34%
Sun 2	4.3	5.8	35%
Digital 5000/25	2.2	2.4	9%
Digital 5000/133	3.2	3.6	13%
Digital 5000/240	4.3	4.7	9%

A comparison of the performance of the two dot-product implementations in figures 2 and 3 is contained in table 1. Note that performance benefits from 9% to 35%, depending upon the architecture, have been obtained.

3.2 Using Scalar Variables

As was discussed briefly in section 2, RISC architectures make extensive use of register variables. For this to be effective, the compiler must handle the assignment of variables to registers. Compilers for RISC architectures tend to retain scalar variables in registers as much as possible.

An example computation that is helpful in understanding the role of registers in execution performance is 1-dimensional convolution. The task is to convolve a large data vector with a five-element mask. In our examples, the convolution results are added to the data previously held in the result vector, so there are five additions and five multiplications to be performed for each result obtained. A straight-forward solution to this problem is contained in figure 4.

Unfortunately, this seemingly simple procedure cannot be compiled with

```

float dot_unrolled (v1, v2, n)
float *v1, *v2;
int n;
{
    float result = 0.0;
    int i;

    /* Unrolled loop performs 8 operations per iteration. */
    for (i = n; (i -= 8) >= 0; )
    {
        result += v1[0] * v2[0];
        result += v1[1] * v2[1];
        result += v1[2] * v2[2];
        result += v1[3] * v2[3];
        result += v1[4] * v2[4];
        result += v1[5] * v2[5];
        result += v1[6] * v2[6];
        result += v1[7] * v2[7];
        v1 += 8;    v2 += 8;
    }

    /* Original loop is retained to handle last few steps. */
    for (i += 8; --i >= 0; )
        result += *v1++ * *v2++;

    return result;
}

```

Figure 3: Loop unrolled dot-product.

```

void convolve_5_normal (result, data, mask, n)
float *result, *data, *mask;
int n;
{
    int i;

    for (i = n; --i >= 0; )
    {
        *result++ += data[0] * mask[0] + data[1] * mask[1] +
            data[2] * mask[2] + data[3] * mask[3] + data[4] * mask[4];
        data++;
    }
}

```

Figure 4: A straight-forward convolution implementation.

```

void convolve_5_reg (result, data, mask, n)
float *result, *data, *mask;
int n;
{
    int i;
    float m0, m1, m2, m3, m4;

    m0 = mask[0]; m1 = mask[1]; m2 = mask[2]; m3 = mask[3];
    m4 = mask[4];

    for (i = n; --i >= 0; )
    {
        *result++ += data[0] * m0 + data[1] * m1 + data[2] * m2 +
            data[3] * m3 + data[4] * m4;
        data++;
    }
}

```

Figure 5: Using scalar variables to encourage register use.

```

main ()
{
  float result[20], data[20];
  ...
  convolve_5_normal (result, data, result+2, 20);
}

```

Figure 6: A procedure call that causes aliasing.

Table 2: Convolution performance in Mflop (percentage improvement).

Architecture	Normal	Scalars	Unrolled	Both	Fig. 9
Sun 1+/IPC	3.7	3.9 (5)	3.8 (3)	4.1 (11)	5.6 (51)
Sun ELC	5.9	6.3 (7)	6.2 (5)	6.6 (12)	9.0 (53)
Sun 2	7.5	7.8 (4)	7.7 (3)	8.1 (8)	11.3 (51)
Digital 5000/25	5.1	5.5 (8)	5.0 (-2)	5.3 (4)	5.9 (16)
Digital 5000/133	7.0	7.5 (7)	6.8 (-3)	7.3 (4)	8.0 (14)
Digital 5000/240	8.8	9.5 (8)	8.6 (-2)	9.2 (5)	10.1 (15)

maximum efficiency because of hidden subtleties that result from the C language definition. In C, it is possible that the arrays pointed to by `result`, `data` and `mask` may be the same data array, or may partially overlap. This situation is called *aliasing*, where different variable names actually refer to the same underlying data object. For example, if `convolve_5_normal` was called by the program fragment in figure 6, then `result[2]` would be an alias for `mask[0]`.

The possibility of aliasing means that the compiler must assume that any of the `mask` or `data` values could be changed every time a `result` is updated. Thus, there is no guarantee that the five mask values are the same in each and every loop iteration. Instead of assigning these values to registers, it loads them from memory in each loop iteration, just in case they have changed. This introduces an overhead that can be eliminated simply by putting the mask values into scalar variables as in figure 5. The performance benefits achieved by this simple modification are modest, as shown in table 2.

3.3 Combining the Techniques

The previous two subsections have presented two methods of obtaining performance improvement: loop unrolling and employing scalar variables. Although the application of scalar variables on its own achieved only a marginal performance improvement, in combination with loop unrolling, much more sig-

```

void convolve_5_unrolled (result, data, mask, n)
float *result, *data, *mask;
int n;
{
    int i;

    for (i = n; (i -= 8) >= 0; )
    {
        result[0] += data[0] * mask[0] + data[1] * mask[1] +
            data[2] * mask[2] + data[3] * mask[3] + data[4] * mask[4];
        result[1] += data[1] * mask[0] + data[2] * mask[1] +
            data[3] * mask[2] + data[4] * mask[3] + data[5] * mask[4];
        result[2] += data[2] * mask[0] + data[3] * mask[1] +
            data[4] * mask[2] + data[5] * mask[3] + data[6] * mask[4];
        result[3] += data[3] * mask[0] + data[4] * mask[1] +
            data[5] * mask[2] + data[6] * mask[3] + data[7] * mask[4];
        result[4] += data[4] * mask[0] + data[5] * mask[1] +
            data[6] * mask[2] + data[7] * mask[3] + data[8] * mask[4];
        result[5] += data[5] * mask[0] + data[6] * mask[1] +
            data[7] * mask[2] + data[8] * mask[3] + data[9] * mask[4];
        result[6] += data[6] * mask[0] + data[7] * mask[1] +
            data[8] * mask[2] + data[9] * mask[3] + data[10] * mask[4];
        result[7] += data[7] * mask[0] + data[8] * mask[1] +
            data[9] * mask[2] + data[10] * mask[3] +
            data[11] * mask[4];
        data += 8;    result += 8;
    }

    for (i += 8; --i >= 0; )
    {
        *result++ += data[0] * mask[0] + data[1] * mask[1] +
            data[2] * mask[2] + data[3] * mask[3] + data[4] * mask[4];
        data++;
    }
}

```

Figure 7: Unrolled convolution implementation.

```

void convolve_5_ureg (result, data, mask, n)
float *result, *data, *mask;
int n;
{
    int i;
    float m0, m1, m2, m3, m4;

    m0 = mask[0]; m1 = mask[1]; m2 = mask[2]; m3 = mask[3];
    m4 = mask[4];

    for (i = n; (i -= 8) >= 0; )
    {
        result[0] += data[0] * m0 + data[1] * m1 + data[2] * m2 +
            data[3] * m3 + data[4] * m4;
        result[1] += data[1] * m0 + data[2] * m1 + data[3] * m2 +
            data[4] * m3 + data[5] * m4;
        result[2] += data[2] * m0 + data[3] * m1 + data[4] * m2 +
            data[5] * m3 + data[6] * m4;
        result[3] += data[3] * m0 + data[4] * m1 + data[5] * m2 +
            data[6] * m3 + data[7] * m4;
        result[4] += data[4] * m0 + data[5] * m1 + data[6] * m2 +
            data[7] * m3 + data[8] * m4;
        result[5] += data[5] * m0 + data[6] * m1 + data[7] * m2 +
            data[8] * m3 + data[9] * m4;
        result[6] += data[6] * m0 + data[7] * m1 + data[8] * m2 +
            data[9] * m3 + data[10] * m4;
        result[7] += data[7] * m0 + data[8] * m1 + data[9] * m2 +
            data[10] * m3 + data[11] * m4;
        data += 8;    result += 8;
    }

    for (i += 8; --i >= 0; )
    {
        *result++ += data[0] * m0 + data[1] * m1 + data[2] * m2 +
            data[3] * m3 + data[4] * m4;
        data++;
    }
}

```

Figure 8: Unrolled convolution with scalars for the mask.

nificant improvements can be obtained. In order to demonstrate this, consider once again the convolution program of figure 4. Simple loop unrolling without attention to scalar variables (figure 7) yields a small performance improvement comparable to that achieved by employing scalar variables without loop unrolling (see table 2). Combining both techniques in the straight-forward way (figure 8) yields an improvement equal to the sum of the individual improvements.

However, loop unrolling opens up many more opportunities for employing scalar variables. Firstly, there is repeated use of `data[4]`, `data[5]`, etc. within the unrolled loop. Although these references are within the same loop iteration, the compiler does not assume that the same data values are involved and so generates multiple load instructions for each data value. This is not unreasonable because the C language definition allows the possibility that the `data` array may be aliased with the `results` array so that the result computed in one convolution statement may affect the data used for the next convolution. Because of this possibility, the compiler is forced to reload the elements of the `data` array each time they are referenced, just in case they have been changed. For the same reason, the compiler has to schedule the instructions so that the computation of each convolution result is completed before starting the computation of the next convolution result. Since floating-point multiplications take several instruction cycles, unfavourably scheduled instructions may result in the CPU stalling to wait for an operation to complete. When the program is modified to use scalar variables `d4`, `d5`, etc., there is no longer any dependence upon changes in the value of `data[4]`, `data[5]`, etc. and the compiler is then more free to schedule the instructions optimally as well as avoiding loading the values from the `data` array more than once each.

Secondly, we observe that `data[8]` near the end of the loop is an alias for `data[0]` at the beginning of the loop in the next loop iteration. Similarly, `data[9]` is an alias for `data[1]`, etc. By using scalar variables `d0`, `d1`, etc. to hold these values, we can prevent the compiler generating code to reload them from memory (see figure 9). The performance improvement achieved with this final version of the convolution program is indeed significant at around 50% (see table 2).

4 Consider the Cache

The SPARCstation and Digital 5000 range of computers employ a cache memory to improve memory access time. The use of cache memory in computer systems is common, but the impact of cache performance on program execution is rarely considered by programmers. The cache in the workstations we are considering is direct-mapped; i.e. if two different addresses share the same cache location, the cache cannot retain data from the two different locations at the one time. This limitation can lead to conflicts in the use of the cache within the operation

```

void convolve_5_maxreg (result, data, mask, n)
float *result, *data, *mask;
int n;
{
    int i;
    float m0, m1, m2, m3, m4;
    float d0, d1, d2, d3, d4, d5, d6, d7;

    m0 = mask[0]; m1 = mask[1]; m2 = mask[2]; m3 = mask[3];
    m4 = mask[4];
    d0 = data[0]; d1 = data[1]; d2 = data[2]; d3 = data[3];

    for (i = n; (i -= 8) >= 0; )
    { d4 = data[4];
      result[0] += d0 * m0 + d1 * m1 + d2 * m2 + d3 * m3 + d4 * m4;
      d5 = data[5];
      result[1] += d1 * m0 + d2 * m1 + d3 * m2 + d4 * m3 + d5 * m4;
      d6 = data[6];
      result[2] += d2 * m0 + d3 * m1 + d4 * m2 + d5 * m3 + d6 * m4;
      d7 = data[7];
      result[3] += d3 * m0 + d4 * m1 + d5 * m2 + d6 * m3 + d7 * m4;
      d0 = data[8];
      result[4] += d4 * m0 + d5 * m1 + d6 * m2 + d7 * m3 + d0 * m4;
      d1 = data[9];
      result[5] += d5 * m0 + d6 * m1 + d7 * m2 + d0 * m3 + d1 * m4;
      d2 = data[10];
      result[6] += d6 * m0 + d7 * m1 + d0 * m2 + d1 * m3 + d2 * m4;
      d3 = data[11];
      result[7] += d7 * m0 + d0 * m1 + d1 * m2 + d2 * m3 + d3 * m4;
      data += 8; result += 8;
    }

    for (i += 8; --i >= 0; )
    { *result++ += data[0] * m0 + data[1] * m1 + data[2] * m2 +
      data[3] * m3 + data[4] * m4;
      data++;
    }
}

```

Figure 9: Unrolled convolution with maximum scalars.

of a program, with detrimental effect on the program performance.² For large regular computations, such as image processing, cache conflicts can degrade program performance as much as five times.

4.1 Measuring Cache Parameters

Four important parameters of cache design are the associativity, the size of the cache, the cache line width, and the cache miss penalty. The set associativity of a cache indicates the number of data that can be cached at a single cache location simultaneously. A direct-mapped cache has set associativity of one. The size of the cache is simply the amount of cache memory available. The cache line width is the amount of data that is transferred to the cache as the result of a cache miss. This data is then available for subsequent accesses, increasing the likelihood of a cache hit. The cache miss penalty is the additional time cost of a cache miss resulting from the necessity of accessing main memory. Cache miss penalties may differ for read and write accesses; this report only considers the cache miss penalty on read accesses. An additional design decision of importance is whether memory write operations affect the cache or write around it. Write around caches can avoid conflicts when copying data between conflicting addresses because the cache is not involved in the write operation.

User programs may also be affected by whether the cache is a virtual address cache or a physical address cache. The data in a virtual address cache is organised on the basis of the program virtual address, so cache conflicts between program addresses will be consistent. The data in a physical address cache is stored on the basis of the physical memory address, which varies depending upon the assignment of virtual addresses to physical addresses by the operating system. If the page size is smaller than the cache size, then cache conflicts between program addresses may vary between invocations of a program, and even within the life of the program as a result of paging activity. Virtual address caches are preferred for speed while physical address caches are preferred for simplicity of the operating system.

A program (see appendix A) can be used to determine the cache parameters based upon performance measurements. The algorithm detects cache conflicts that occur when two or more different addresses share the same cache location. The conflicting use of the cache reduces performance and allows the cache parameters to be measured. The program assumes that the data cache is a virtual address cache and may produce inconsistent results if it is run on a machine that has a physical address cache.

For a direct-mapped cache, a simple computation such as adding one array to another suffices to obtain cache conflicts. To determine the cache size, the program alternately accesses two arrays that are separated in address space by c bytes. When c is equal to the cache size, a severe performance penalty is

²For a discussion of cache design issues, see [1, chapter 8].

incurred because the access to the first array conflicts with the access to the second array: the cache is unable to store both the array data values, so each access becomes a cache miss, causing main memory to be accessed. The program detects the onset of cache conflict, based upon performance measurements, and thus determines the cache size. It assumes that the cache size is a power of two.

The program determines the set associativity of the cache by starting with the assumption that it is testing a direct-mapped cache. If no conflicts are detected with this assumption, the program tries to obtain conflicts for a 2-way set associative cache by iteratively accessing three arrays, each separated from the other by the hypothesized cache size c . The assumed set associativity of the cache is incrementally increased until cache conflicts are detected. The program is capable of testing for set associativity up to eight.

Once the cache size has been determined, the cache line size can be estimated. To determine the cache line size, the program alternately accesses two arrays that are separated by $c + l$ bytes, where c is the known cache size, and l is the hypothesized cache line size. When l is equal to or larger than the cache line size (but smaller than the cache size), no cache conflicts occur. The program starts by assuming a large power of two for l and repeatedly halves it until a performance penalty is detected. The performance penalty is assumed to result from cache conflicts occurring because l is less than the cache line size. The smallest value of l that provides efficient performance is reported as the cache line size.

The program also determines whether cache misses on write operations are handled by the *fetch on write* or the *write around* strategy. A fetch on write cache will fetch the data from memory into the cache when the write location is not already cached. This strategy is beneficial if the data is likely to be read back in the near future; e.g. if memory is being used to temporarily store a variable. Conversely, the write around strategy handles a cache miss on write by directly accessing the memory. This approach is beneficial when the write would otherwise be in conflict with data reads; e.g. when copying data between two arrays at conflicting addresses.

I have employed this program to measure the cache parameters of a number of RISC workstation architectures. The results are displayed in table 3. The results for the Sun SPARC workstations are highly consistent between executions of the program, which is consistent with a virtual address cache. (A virtual address cache is described for the SPARCstation 1+ in [2]). The cache miss penalty measurement for the Digital machines varies between executions, which may be indicative of a physical address cache.

4.2 Cache Performance Impact

The simple functions used previously to measure the performance impact of loop unrolling can also be used to measure the performance impact of cache conflicts. Recall that these functions compute the dot-product of two floating-

Table 3: Measured cache parameters for RISC workstation architectures.

Architecture	Assoc.	Size	Line	Write Around	Miss Penalty
Sun 1+/IPC	1	65536	16	no	520ns
Sun ELC	1	65536	32	no	765ns
Sun 2	1	65536	32	no	635ns
DEC 5000/25	1	65536	16	yes	1170ns
DEC 5000/133	1	131072	32	yes	1480ns
DEC 5000/240	1	65536	32	yes	1055ns

point arrays. The main program³ allows the gap between the two data arrays to be specified. If the gap is a multiple of the cache size, cache conflicts occur and a performance penalty is paid.

Figures 10 through 13 show the measured dot-product computation performance of three different RISC architectures as a function of the gap between the two data arrays. The gap used in these experiments was close to 2^{18} which is a power of two sufficiently large to produce cache conflicts on all the architectures considered. The performance is graphed for both the normal dot-product computation (figure 2) and the unrolled computation (figure 3). Figures 10 and 11 show the performance of the Sun SPARCstation 1+/IPC and Sun SPARCstation 2 architectures. The performance of the Sun SPARCstation ELC architecture is a scaled version of figure 11. Figures 12 and 13 show the performance of the Digital workstations models 5000/25 and 5000/240 respectively. The Digital model 5000/133 performs similarly to the 5000/240, but the performance is again scaled down.

All of these performance graphs clearly display the severe performance degradation introduced by cache conflicts—up to a factor of five for the faster machines. The effect of the cache line can also be seen. The performance hole is narrow for the architectures with 16-byte cache lines (the Sun SPARCstation 1+ and Digital 5000/25), and wider for the architectures with 32-byte cache lines. Because of this, the slower machines actually outperform the faster machines when the gap between the data arrays is $2^{18} \pm 16$.

5 Solving Cache Conflicts

As demonstrated in figures 10 through 13, cache conflicts can produce severe performance penalties, with performance losses as great as a factor of five. Although computations that are more complex than the dot-product are likely

³All programs discussed in this report are available from the author by electronic mail, or by anonymous FTP from ftp.mpce.mq.edu.au.

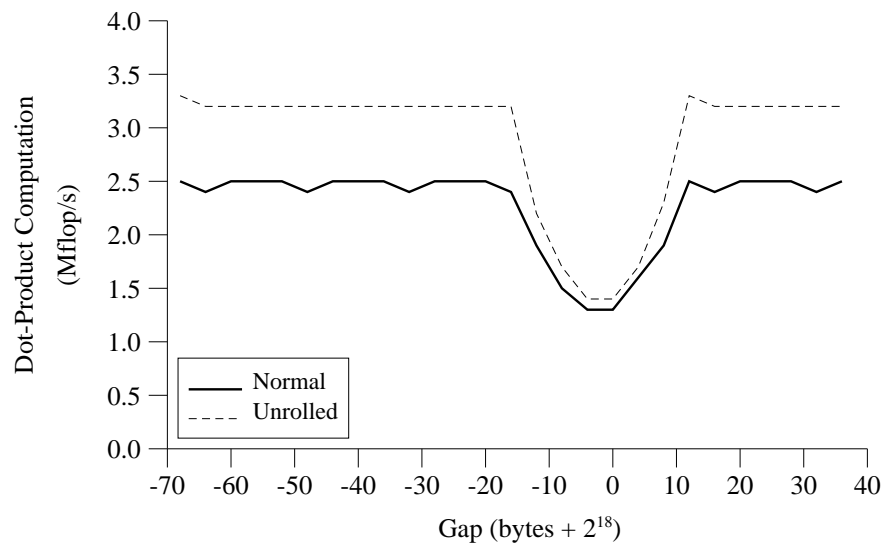


Figure 10: Impact of cache conflicts: Sun SPARCstation 1+/IPC.

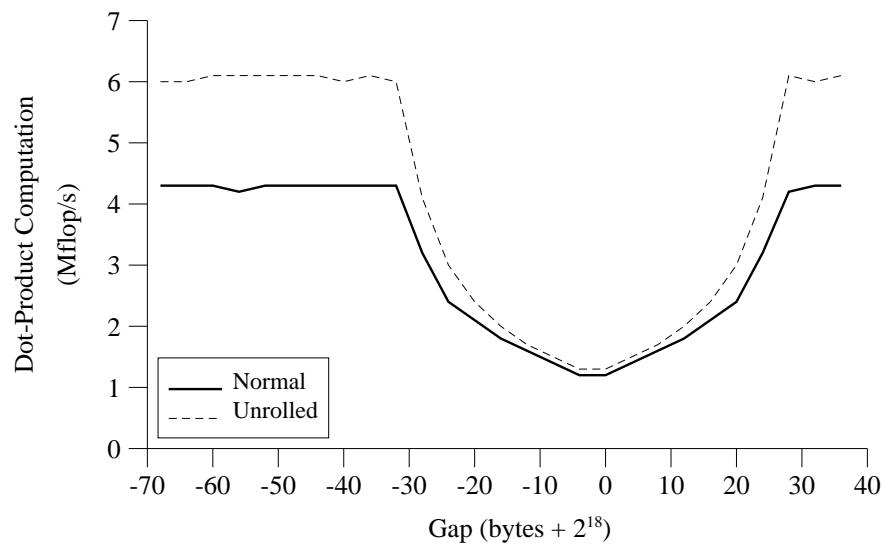


Figure 11: Impact of cache conflicts: Sun SPARCstation 2.

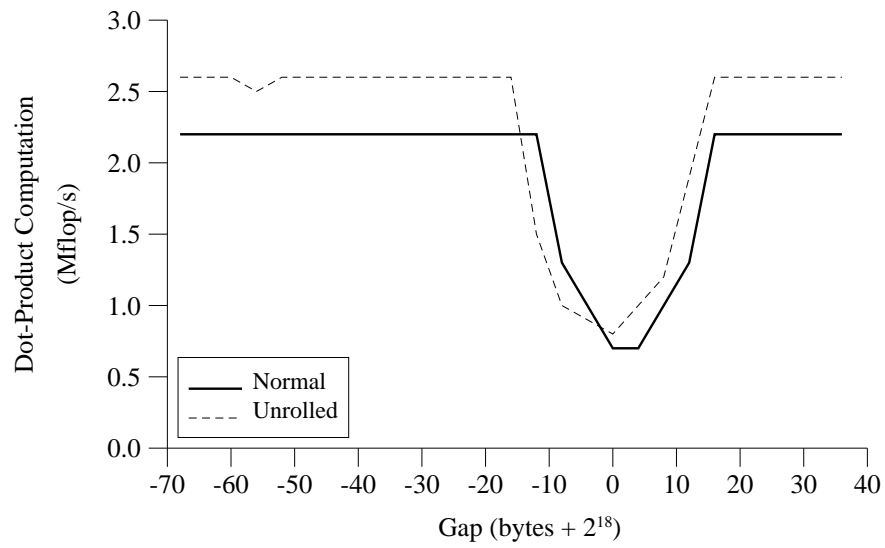


Figure 12: Impact of cache conflicts: Digital 5000/25.

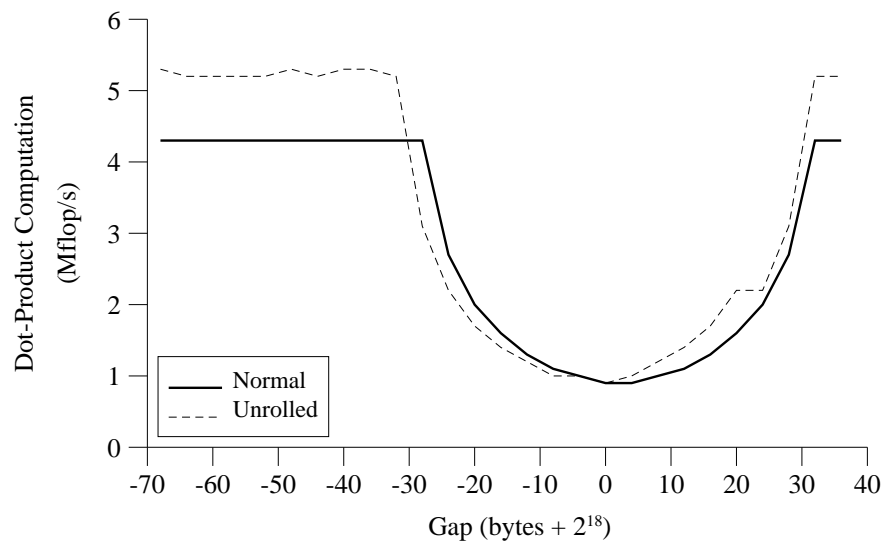


Figure 13: Impact of cache conflicts: Digital 5000/240.

to be less severely penalised by cache conflicts, the possible costs cannot be ignored in large scientific computations. Potentially, any operation involving a one-to-one mapping between arrays could lead to cache conflicts on architectures with a direct-mapped cache. Whether or not conflicts are observed depends largely upon the relative positions of the two arrays, which is usually given no consideration at all in scientific programming.

The onset of cache conflict may occur seemingly unpredictably, resulting from a minor change in some part of a program which affects the memory locations of objects. Suddenly, program performance degrades without any obvious explanation! It may also be the case that a particular program is always experiencing cache conflict problems, either because the arrays are declared in such a way that the compiler always places them in conflicting locations, or because of the operation of the memory allocator. In this case, the full potential of the machine is never realised.

The following subsections discuss a number of approaches that may be employed to avoid or reduce the problem of cache conflict. Because the memory allocator is often used to create the large data arrays used in scientific programs, the operation of the memory allocator can have significant impact on the problem of cache conflicts within a program. Section 5.1 discusses the characteristics of memory allocators and the implications for scientific programming.

Three software solutions to the problem of cache conflicts are then proposed in sections 5.2 through 5.5. In the first solution, the locations of data arrays are randomised, probabilistically reducing the occurrence of cache conflicts. In the second approach, the locations of data arrays are analytically optimised, eliminating cache conflicts entirely. The third approach involves rearranging the data array cache accesses to use bulk data transfers, effectively employing the RISC registers as an on-chip cache. The effectiveness of this technique may, however, be limited by the compiler.

The last solution proposed is a hardware modification in which consistent cache conflict access patterns are prevented. This approach is discussed in section 5.6.

5.1 Importance of the Memory Allocator

Cache conflicts become a problem when large regular computations (such as image processing operations) must be performed on data objects that are in conflict. Because of the regular nature of the computation, cache conflicts that occur will appear regularly throughout the entire computation. Large data arrays for such computations are often obtained by calling the system memory allocator. The potential for cache conflicts will therefore depend upon the design of the memory allocator.

As a worst case, memory allocators based upon the buddy system allocate memory blocks on boundaries that are multiples of the next power of two larger than the requested data block. For large arrays, this approach ensures that

corresponding data elements will be in conflicting locations, since the cache size is also usually a power of two. Fortunately, the buddy system does not appear to be used on any of the RISC workstations considered in this report. However, the risk of cache conflict still exists.

The memory allocator on the Sun systems initially creates objects sequentially in memory with eight bytes of overhead data between them. This means that, if the objects' size is a multiple of the cache size, sequentially allocated objects will be located within eight bytes of conflicting locations. Since the cache line is sixteen or thirty-two bytes, cache conflict is a serious potential problem in programs that create two or more large data objects sequentially. However, if the large data objects are created at different times within the execution of the program, and other objects are created in between, the chance of serious conflict between the large data objects is small.

The memory allocator on the Digital systems also creates objects sequentially in memory, but large objects are aligned to 4k byte boundaries. Thus, if the objects' size is a multiple of the cache size, sequentially allocated objects will avoid cache conflicts by 4k bytes. However, two large objects allocated at different times within the execution of a program have a 1 in 16 chance of being in direct conflict on the DEC 5000/25 (which has a 64k byte cache) and a 1 in 32 chance on the DEC 5000/133 (which has a 128k byte cache).

The potential for cache conflicts resulting from the location of data objects can be handled in two distinct ways. The simplest technique, described in section 5.2, is to add a simple layered procedure to the memory allocator to randomise the memory location of allocated objects. This ensures that whether objects are allocated successively or at disparate times within the life of a program, the chance of direct cache conflict is kept statistically small. The disadvantage of location randomisation is that it is unpredictable—an unfortunate choice of the seed for the random number generator may cause a significant reduction in program performance due to the sudden introduction of a cache conflict between two large data arrays. This problem can be effectively removed in image processing applications by employing an Iliffe vector representation and randomising the Iliffe vector, as described in section 5.3.

As an alternative to randomisation, the locations of objects may be optimised at run time. This technique is much more difficult to implement as it requires computing desirable locations for data objects based upon the computations to be performed on them. Section 5.4 presents an example in which this technique is applied. The advantage of location optimisation is, of course, that it is predictable and can reliably prevent cache conflicts.

5.2 Randomisation of Location

Cache conflicts become a problem in scientific software, especially image processing systems, when two large data arrays must be processed such that there is a consistent pattern of cache conflicts between the data elements being pro-

cessed. The problem can be reduced in significance, then, by attempting to ensure that cache conflicts are an unlikely event. For computations involving vectors, such as the example dot-product and convolution computations discussed in this report, the addresses of the individual vectors can be randomly modified so as to reduce the chance of cache conflict to the statistical minimum.

A technique for performing this randomisation is presented in figure 14 where a simple randomising routine is wrapped around the system memory allocation routine, and a corresponding modification is wrapped around the system memory deallocator. From the point of view of the software engineer, the greatest inconvenience imposed by this approach is the requirement of correctly matching `ran_malloc` and `ran_free` calls, since `ran_malloc` is incompatible with `free` and `ran_free` is incompatible with `malloc`. Of course, there is no guarantee that any two large vectors will not happen to be assigned conflicting addresses. If the same vectors must be repeatedly processed, a random conflict could still dominate the overall system performance. The occurrence of cache conflict in a program may be detected, however, by varying the seed for the random number generator so that different addressing patterns are tried.

5.3 Iliffe Vector Randomisation

The problem of randomly generated conflicts can be mitigated in image processing systems because these systems mostly operate upon two-dimensional image data. Images can be efficiently stored using the Iliffe vector approach, in which the two-dimensional array is represented by a vector of pointers, each of which contains the address of a single row vector (see figure 15). In the C language, the Iliffe vector representation is convenient because it can be accessed with a two-dimensional array subscript notation `a[i][j]`. It is also more convenient than a conventional two-dimensional array when passing images between procedures because the conventional array requires explicit subscript computation unless the second dimension is fixed.

From the perspective of avoiding cache conflict problems, a more significant benefit of the Iliffe representation of two-dimensional arrays is the possibility of randomising the memory access patterns between rows. Two randomisation techniques are available. The simplest is to randomise the locations of the individual row vectors in the manner of figure 14. This will effectively prevent cache conflicts from occurring consistently throughout the processing of two or more arrays, but is wasteful of memory.

A second technique for preventing cache conflicts using an Iliffe representation is to shuffle the pointer vector so that successive rows of the two-dimensional array are stored in a random pattern in memory. This technique will not be effective, however, if the addresses of the row vectors are all the same modulo the cache size. Ideally, the row vectors should be several times smaller than the cache, although it is sufficient that their starting addresses should map to several, well separated locations in the cache. In the architectures considered

```

/* MALLOCALIGN must be at least sizeof (char *) */
# define MALLOCALIGN sizeof (double)
# define CACHESIZE (1<<16)

char *ran_malloc (size)
int size;
{
    char *malloc (), *block, **ptr;
    int randomisation = size / 2, adjust;
    long int random ();

    if (randomisation > CACHESIZE)
        randomisation = CACHESIZE;

    /* adjust must be at least 1 to allow for storing a pointer
     * to the beginning of the block. */
    adjust = random () % (randomisation / MALLOCALIGN + 1) + 1;
    block = malloc (size + adjust * MALLOCALIGN);

    ptr = (char **) (block + adjust * MALLOCALIGN);
    ptr[-1] = block;

    return (char *) ptr;
}

char *ran_free (ptr)
char *ptr;
{
    char **p = (char **) ptr;
    free (p[-1]);
}

```

Figure 14: Randomisation of allocated addresses.

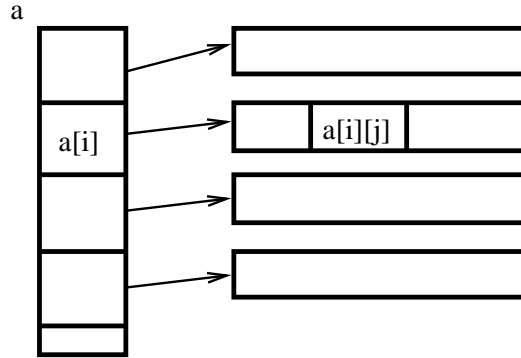


Figure 15: Iliffe representation of a two-dimensional array.

in this report, the cache size is always at least 64k bytes, so it is not difficult to effectively shuffle typical image row vectors.

Shuffling the entire pointer vector may not be advisable because a totally random access pattern may produce inefficient paging activity. A simple solution to this problem is to preserve data locality by shuffling the Iliffe vector in blocks. The routine `iliffe_shuffle` presented in figure 16 shuffles the Iliffe vector by randomly exchanging pairs of pointers. Initially, the blocks consist of single pointers, then the shuffled pointer pairs are themselves shuffled by random exchanges. The resulting blocks of four pointers are then shuffled, and the block size is again doubled. This process continues until the block size exceeds the size of the Iliffe vector. This method preserves the locality of data within every block of pointers of size 2^m on a 2^m boundary (for all m). This ensures that the access pattern is not random, and paging activity is not seriously impacted. For an Iliffe vectors of n elements, representing an image of n rows, there are approximately 2^{n-1} possible shuffled Iliffe vectors, so the risk of coincidental conflict patterns is negligible for reasonable values of n .

Figure 17 shows the results of a simulation of the occurrence of cache conflicts when the Iliffe vector has been shuffled by the `iliffe_shuffle` procedure. The simulation assumes a computation that involves iterative computation over two data arrays. On the Sun architectures, two-dimensional convolution would be an example of such a computation. On the Digital architectures, which have a write-around cache, an operation such as adding two images together would produce the type of cache conflicts simulated here.

In figure 17, the rate of conflict decreases as the reciprocal of the number of non-conflicting row vectors. This performance pattern reflects the requirement that the starting addresses of the row vectors map to well separated locations in the cache. If there are only two distinct starting addresses in relation to the

```

iliffe_shuffle (ptr, nr)
char **ptr;
int nr;
{
    long int random ();
    int i, first, second, end, blocksize;
    char *hold;

    /* Shuffle by randomly exchanging pairs of blocks.
     * Start with block size 1, then 2, 4, 8, ...
     * Preserves data locality while providing effective
     * shuffling.
     */
    for (blocksize = 1; blocksize < nr; blocksize *= 2)
        for (second = blocksize; second < nr; second += blocksize*2)
            {
                /* Exchange pair of blocks with 50% probability */
                if (random () & 1)
                    {
                        /* Swap entire blocks. */
                        first = second - blocksize;
                        end = second + blocksize;
                        if (end > nr)
                            end = nr;
                        for (i = second; i < end; i++, first++)
                            { hold = ptr[first];
                              ptr[first] = ptr[i];
                              ptr[i] = hold;
                            }
                    }
            }
}

```

Figure 16: Procedure to shuffle an Iliffe vector.

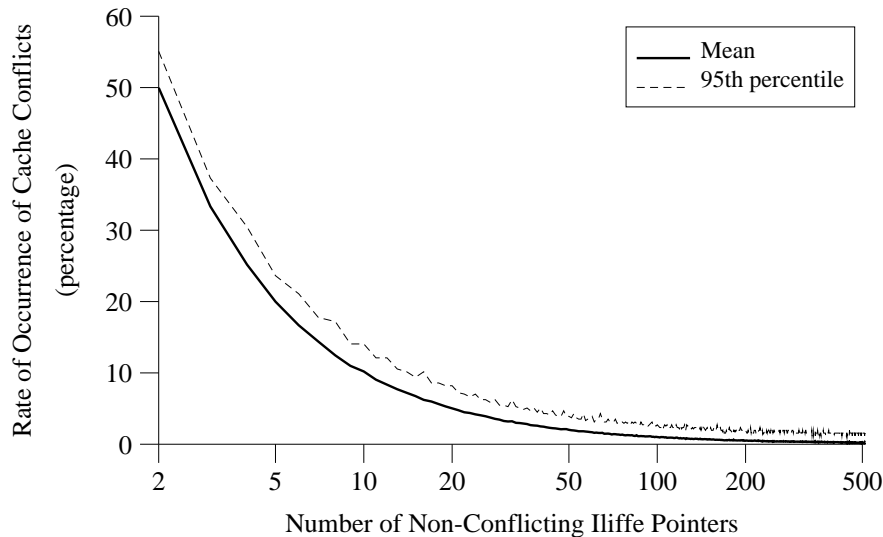


Figure 17: Reduction of cache conflicts by Iliffe vector shuffling.

cache, then conflicts are encountered 50% of the time. This would happen if the row vectors were close to half the size of the cache, or close to 1.5, 2.5, 3.5, etc. times the size of the cache. In such a situation, the simplest solution is to increase the row vector size by a small amount so that many well separated cache locations are addressed. The result will be a significant improvement in the effectiveness of the Iliffe vector shuffling approach.

Figure 17 also shows the 95th percentile of the rate of cache conflicts. This curve provides an indicator of the probabilistic upper limit on the performance of shuffled Iliffe vectors for images of 512 rows. (The performance variability will be greater and hence the probabilistic upper limit will also be greater for Iliffe vectors of less than 512 elements.) If there are 512 well separated row pointers relative to the cache, the 95th percentile of cache conflicts is 1.5% for an image of 512 rows.

The architectures discussed in this report have relatively large caches—64k bytes for most models (128k bytes for the Digital model 5000/133). Such a large cache means that 128 rows of a typical 512×512 byte image or 32 (64) rows of a 512×512 floating-point image can be accommodated simultaneously in the cache. Thus, these image configurations will typically produce cache conflicts no more than 3% of the time which involves a performance penalty of 11%. (The 95th percentile represents a 6% cache conflict rate and a performance penalty of 20%.) An even lower conflict rate will be obtained if the row vectors are enlarged

```

# define LINESIZE 32
# define CACHESIZE (1<<16)

main (ac, av)
int ac;
char **av;
{
    float *x, *y, result;
    int n, conflict;
    char *malloc ();

    /* (Initialise n) */

    x = (float *) malloc (n * sizeof (float));
    y = (float *) malloc (n * sizeof (float) + LINESIZE);
    conflict = (((long int) y) - ((long int) x)) & (CACHESIZE-1);
    if (conflict <= LINESIZE)
        y += LINESIZE / sizeof (float);

    /* (Initialise vectors x and y) */

    result = dot_unrolled (x, y, n);
}

```

Figure 18: Optimisation of location.

slightly (e.g. to 513 bytes or 513 floating-point values) so that there are 512 well separated row vector locations relative to the cache. The cache conflict rate will then be reduced to 0.19% on average (0.7% performance penalty) and 1.56% for the 95th percentile (5.8% performance penalty).

5.4 Optimisation of Location

In situations where large data arrays are to be used in particular computations, it may be possible to compute interdependencies between the arrays and adjust their addresses to prevent cache conflicts in the computations. For example, if two vectors are to be involved in dot-product computations, it is necessary merely to ensure that the starting addresses of the two vectors differ by at least the cache line size. This requirement can be easily satisfied using the technique illustrated in figure 18.

In more complex cases, with several large arrays involved in the computation, some addresses may need to be adjusted by more than one increment of the cache line size. If the computation involves reusing the rows of two-dimensional

arrays (such as occurs with image kernel operations like convolution), then it may be possible to optimise the array locations to prevent cache conflicts not only between the data involved in any one point of the computation, but between all the data involved in an entire row pass. The result of this more stringent optimisation is that the processing of subsequent rows is more efficient because the reused row data is retained in the cache.

I have applied these techniques to the task of convolving an image with a 3×3 mask. Without location optimisation, the program (which was running on a Sun SPARCstation 2) took 0.92 seconds to process a 512×512 floating-point image. When cache conflicts were eliminated, the time was reduced to 0.60 seconds, a saving of 35%. When the optimisation was extended so that the input data involved in consecutive row passes of the convolution was retained in cache, the time was further reduced to 0.54 seconds, a further saving of 10% and an overall saving of 42%.

5.5 Bulk Data Transfers

The previous sections have proposed software solutions to the cache conflict problem that involve modifying the locations of the data arrays. An alternative to this approach is to modify the computation itself so that the pattern of memory access does not generate cache conflict, even when the data arrays themselves are in conflicting locations. The problem of cache conflicts arises because the program accesses a data value (say `v1[0]`) then immediately accesses another data value (say `v2[0]`) which happens to conflict in the cache. The conflict causes the cache line to be replaced and, as a result, when the program subsequently accesses `v1[1]`, it is not found in the cache. Neither is `v2[1]` and so on. The problem of cache conflict, then, arises from two simultaneous conditions.

- Two (or more) data arrays are located so that their cache entries conflict.
- The conflicting data arrays are being accessed in a tightly interleaved pattern.

Thus, a second approach to eliminating cache conflicts is to change the computation itself so that the pattern of memory access is not tightly interleaved between two or more data arrays. Then, even if the arrays are themselves in conflict, the computation will not suffer the severe performance penalties observed in section 4.2.

The elimination of interleaved access patterns starts with a loop-unrolled version of the program, such as the function `dot_unrolled` in figure 3. This code is then modified so that the data arrays are accessed in a non-interleaved manner. For example, in figure 19, the eight values of vector `v1` are loaded into scalar variables before computing the result using the eight values of vector `v2`.

If this program was executed as written, and assuming that the scalar variables were held in registers, the problem of cache conflicts would be largely eliminated.

Unfortunately, function `dot_bulk` does not execute as written. The C compiler, on both the Digital and Sun machines, employs an obvious optimisation and deduces that the loading of `v1[0]` into `v10` can be deferred until `v10` is needed in the computation of the result. Similarly, it defers the loading of `v1[1]`, `v1[2]`, etc., so that the performance of the compiled program is almost identical to that obtained for the ordinary unrolled computation (see figures 23 through 28).

Good results can be obtained, however, by loading data from both vectors into scalars. Rather than load both data arrays ahead of the computation, the data from vector `v2` is loaded after the computation, pre-loading the data for the next loop iteration (see figure 20). The compilers schedule the resulting code so that the load instructions are mixed with the floating-point computation instructions and the performance obtained on the Sun machines is close to that achieved by the unrolled loop when cache conflicts are not occurring, and still very good when the data arrays are at conflicting addresses (see figure 23 through 25).

The performance of `dot_inline` on the Digital workstations when there are no cache conflicts is significantly degraded compared to the performance of `dot_unrolled`. This problem occurs because the compiler allocates some of the scalar variables to that stack—an examination of the assembly code shows the load and store operations to the stack. The likely reason for such an action by the compiler is that the machine's register set has been exhausted. The most reasonable solution available is to reduce the loop unrolling factor. By using a loop unrolling factor of four (figure 21), excellent performance is obtained when the data arrays are not at conflicting addresses, and reasonable performance is still obtained when the data arrays are in conflict (see figures 26 through 28).

For comparison, I also tested a variation of the dot-product function in which the unrolled computation is merged into a single statement (figure 22). This modifies the computation somewhat, as each group of eight vector elements is combined into an individual dot-product before being added to the running result. Although such a modification improves the numerical properties of the computation slightly, it appears to adversely affect the compiler's ability to schedule the instruction mix, and the resulting programs always ran slower than the corresponding routine with separated statements (see figures 23 through 28).

5.6 A Hardware Solution

A simple hardware modification could be employed by vendors to mitigate the problem of cache conflicts in scientific computations. The proposed modification is to hash the high bits of the memory address together with the cache line number thus changing the sequence of the cache lines as the high bits of the memory address change. The simplest hashing method would be to combine

```

float dot_bulk (v1, v2, n)
float *v1, *v2;
int n;
{
    float result = 0.0;
    int i;
    float v10, v11, v12, v13, v14, v15, v16, v17;

    for (i = n; (i -= 8) >= 0; )
    {
        v10 = v1[0];  v11 = v1[1];
        v12 = v1[2];  v13 = v1[3];
        v14 = v1[4];  v15 = v1[5];
        v16 = v1[6];  v17 = v1[7];
        result += v10 * v2[0];
        result += v11 * v2[1];
        result += v12 * v2[2];
        result += v13 * v2[3];
        result += v14 * v2[4];
        result += v15 * v2[5];
        result += v16 * v2[6];
        result += v17 * v2[7];
        v1 += 8;      v2 += 8;
    }

    for (i += 8; --i >= 0; )
        result += *v1++ * *v2++;

    return result;
}

```

Figure 19: Bulk loading of scalars in an attempt to reduce cache conflicts.

```

float dot_inline (v1, v2, n)
float *v1, *v2;
int n;
{
    float result = 0.0;
    int i;
    float v10, v11, v12, v13, v14, v15, v16, v17;
    float v20, v21, v22, v23, v24, v25, v26, v27;

    v20 = v2[0];    v21 = v2[1];
    v22 = v2[2];    v23 = v2[3];
    v24 = v2[4];    v25 = v2[5];
    v26 = v2[6];    v27 = v2[7];

    for (i = n; (i -= 8) >= 0; )
    {
        v10 = v1[0];    v11 = v1[1];
        v12 = v1[2];    v13 = v1[3];
        v14 = v1[4];    v15 = v1[5];
        v16 = v1[6];    v17 = v1[7];
        result += v10 * v20; result += v11 * v21;
        result += v12 * v22; result += v13 * v23;
        result += v14 * v24; result += v15 * v25;
        result += v16 * v26; result += v17 * v27;
        v20 = v2[8];    v21 = v2[9];
        v22 = v2[10];   v23 = v2[11];
        v24 = v2[12];   v25 = v2[13];
        v26 = v2[14];   v27 = v2[15];
        v1 += 8;        v2 += 8;
    }

    for (i += 8; --i >= 0; )
        result += *v1++ * *v2++;

    return result;
}

```

Figure 20: Inline loading of scalars to reduce cache conflicts.

```

float dot_inline_4 (v1, v2, n)
float *v1, *v2;
int n;
{
    float result = 0.0;
    int i;
    float v10, v11, v12, v13;
    float v20, v21, v22, v23;

    v20 = v2[0];    v21 = v2[1];
    v22 = v2[2];    v23 = v2[3];

    for (i = n; (i -= 4) >= 0; )
    {
        v10 = v1[0];    v11 = v1[1];
        v12 = v1[2];    v13 = v1[3];
        result += v10 * v20; result += v11 * v21;
        result += v12 * v22; result += v13 * v23;
        v20 = v2[4];    v21 = v2[5];
        v22 = v2[6];    v23 = v2[7];
        v1 += 4;        v2 += 4;
    }

    for (i += 4; --i >= 0; )
        result += *v1++ * *v2++;

    return result;
}

```

Figure 21: Inline loading of scalars, four times unrolled.

```

float dot_unrolled_merged (v1, v2, n)
float *v1, *v2;
int n;
{
    float result = 0.0;
    int i;

    /* Unrolled loop performs 8 operations per iteration. */
    for (i = n; (i -= 8) >= 0; )
    {
        result += v1[0] * v2[0] + v1[1] * v2[1] + v1[2] * v2[2] +
            v1[3] * v2[3] + v1[4] * v2[4] + v1[5] * v2[5] +
            v1[6] * v2[6] + v1[7] * v2[7];
        v1 += 8;    v2 += 8;
    }

    /* Original loop is retained to handle last few steps. */
    for (i += 8; --i >= 0; )
        result += *v1++ * *v2++;

    return result;
}

```

Figure 22: Merged dot-product computation.

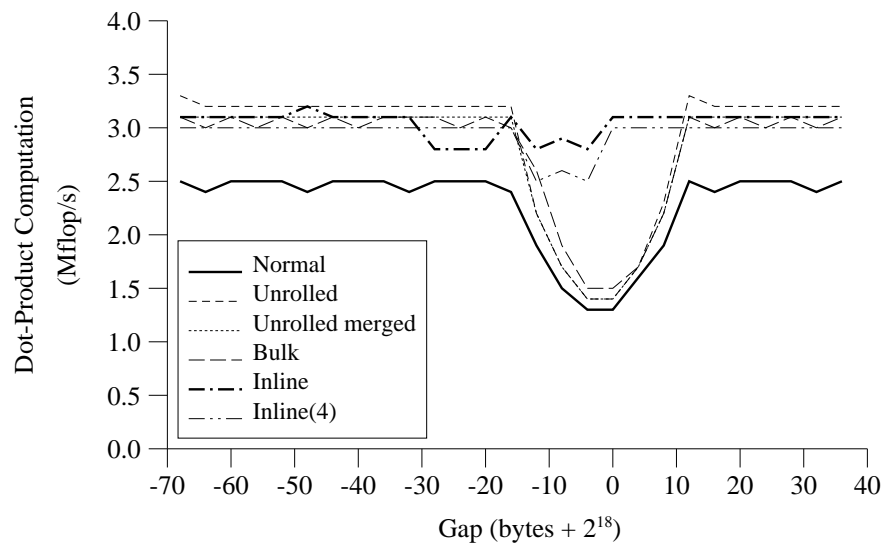


Figure 23: Performance of dot-product on Sun SPARCstation 1+/IPC.

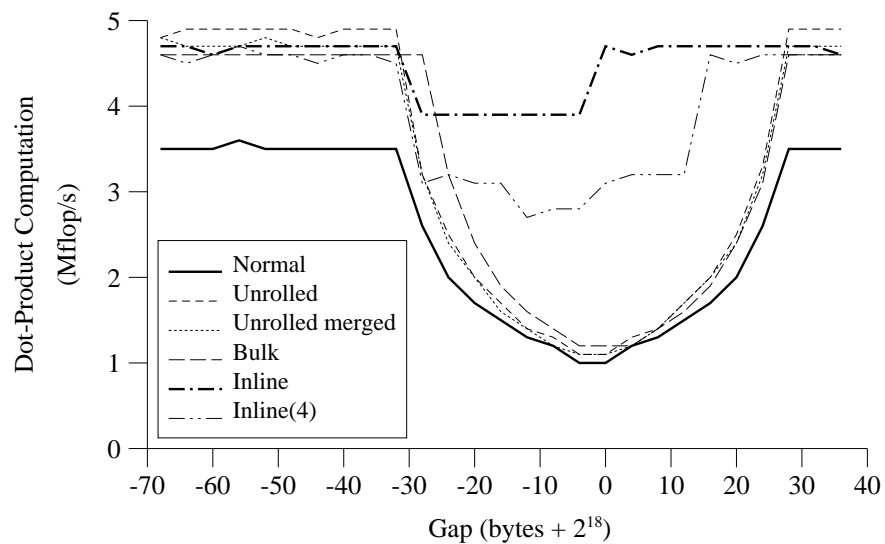


Figure 24: Performance of dot-product on Sun SPARCstation ELC.

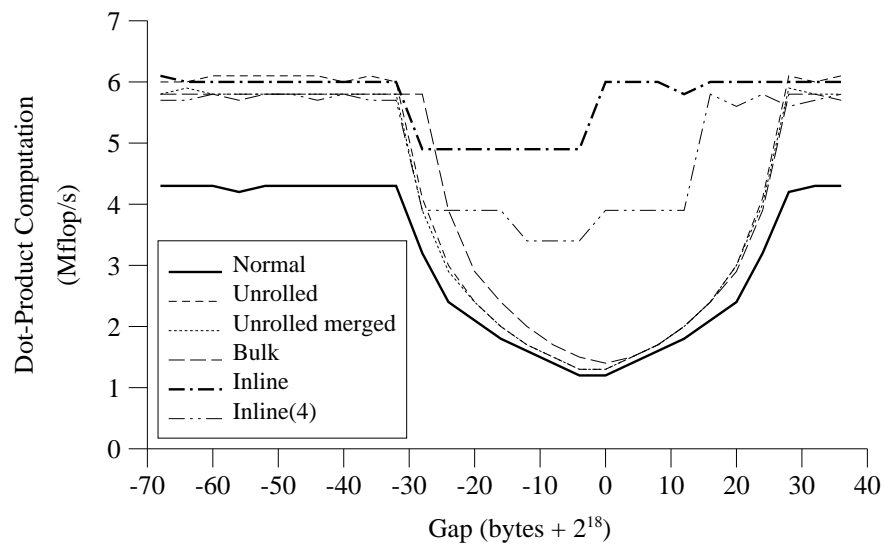


Figure 25: Performance of dot-product on Sun SPARCstation 2.

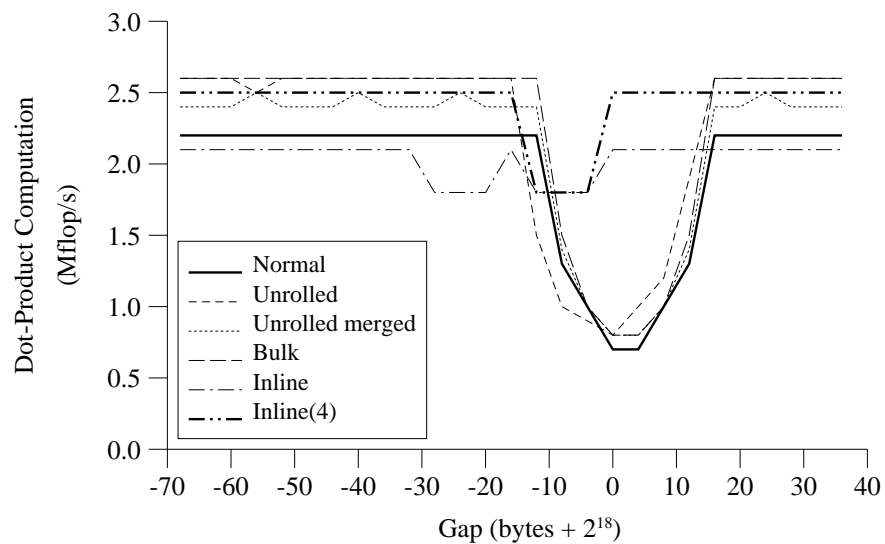


Figure 26: Performance of dot-product on Digital 5000/25.

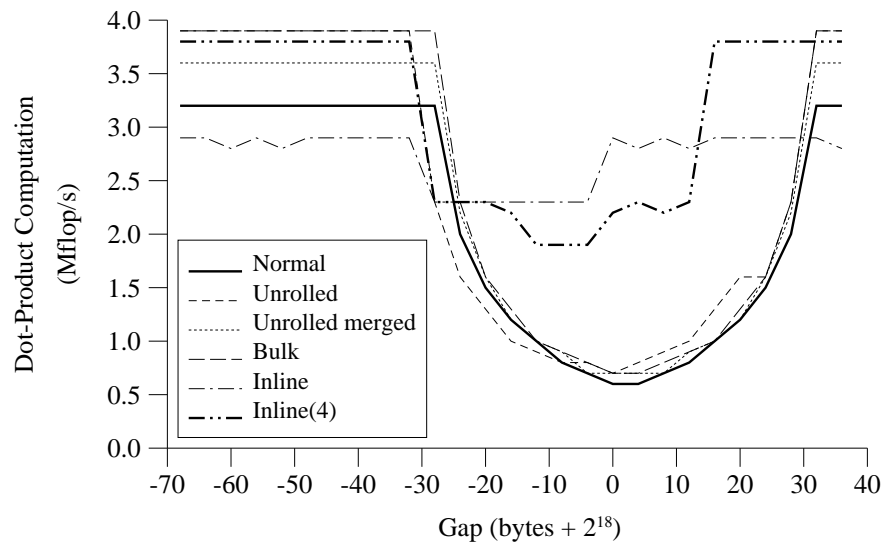


Figure 27: Performance of dot-product on Digital 5000/133.

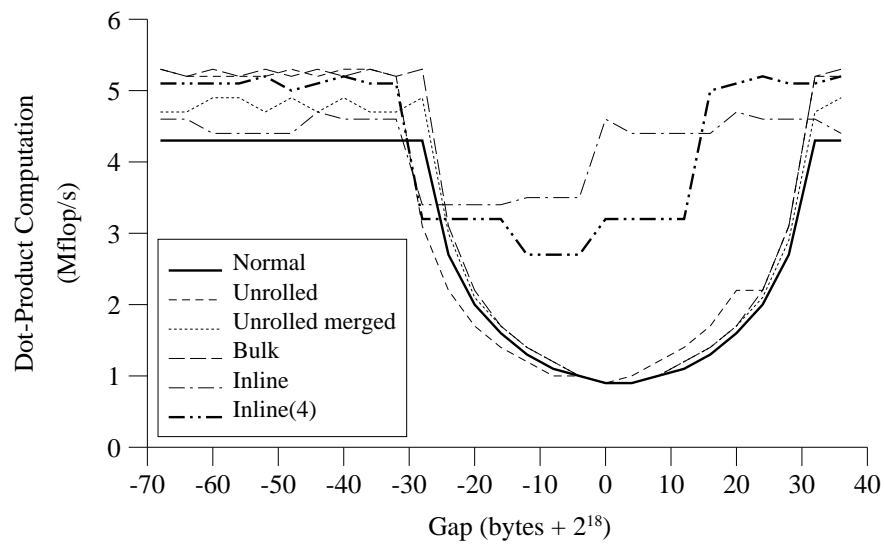


Figure 28: Performance of dot-product on Digital 5000/240.

the bits by exclusive-or. This would mean that, if the addresses of two arrays differed in only one of the high bits, their cache usage patterns would differ such that, in the worst case, cache conflicts would occur at most 50% of the time and the cache conflict penalty would be halved. More sophisticated hashing schemes could be even more effective at reducing the incidence of cache conflicts.

Of course, cache conflict problems can be eliminated by employing a set-associative data cache. The technique of cache address hashing proposed here provides a low-cost alternative by enhancing the direct-mapped cache design.

6 Prognosis

The architectures studied in this report are not the most recently announced machines. Rather, they are machines in everyday use for image processing work and research. New machines are regularly announced and some of the latest architectures appear to offer significant improvements for image processing. The improvements to be gained through manual loop unrolling will likely be reduced in the future through improved compiler technology. The benefits of scalar variable assignment, however, derive largely from the semantic limitations of the programming language. These limitations will remain in the current languages and, as a result, the benefits of scalar variable assignment are likely to remain.

The prognosis for cache conflict problems is mixed. The new SuperSPARC chip [3] has a 4kb on-chip data cache. This cache is four-way set associative, which will eliminate cache conflict problems for computations involving up to four data objects simultaneously. Thus, the dot-product and convolution computations discussed in this paper can be expected to perform consistently irrespective of the relative addresses of the data arrays involved. The implication is that some existing scientific software applications, those that are currently experiencing cache conflicts, will exhibit a greater performance improvement when moved to a SuperSPARC architecture than would be expected purely on the basis of the relative speed ratings of the machines.

It is worth noting that an on-chip set-associative cache does not need to be large to address the problem of cache conflicts discussed in this report. The SuperSPARC chip also supports a large (1Mb) external cache which will address the caching of rows of images. However, even if cache conflicts occurred in this external cache, the impact upon the overall execution time would be much less severe than the factor of five observed in section 4.2 because the on-chip cache will prevent fine-grained cache conflicts.

Other new architectures may continue to employ direct-mapped caches as their first-level caches. The potential for cache conflict problems will then remain. When selecting machines for image processing applications, researchers would therefore be advised to consider not only the benchmark performance of the machine, but also its cache design.

7 Conclusion

The computational performance of image processing programs depends not only upon the raw power of the machine, but also upon the design on the program. There are many issues in designing an efficient program—some relate to the selection of an efficient algorithm, and some relate to the implementation of that algorithm efficiently on the underlying computer architecture.

Two techniques for improving program performance, largely independent of the precise computation involved, are looping unrolling and employing scalar variables. Loop unrolling increases the amount of useful computation performed in each loop iteration. The usefulness of this technique depends upon architectural considerations and also upon compiler technology, since compilers may perform loop unrolling automatically.

The generous register set of typical RISC architectures may be beneficially employed by loading data from arrays and structured variables into simple scalar variables. The effectiveness of this technique is largely due to semantic limitations in the high-level language which necessitate the compiler generating redundant load operations and place restrictions upon the scheduling of instructions by the compiler. When both the loop unrolling and scalar variable techniques are combined, very significant performance improvements can be achieved, even on more complex computations such as convolution.

Another characteristic of the computer architecture, the system cache, may also significantly impact the performance of image processing programs. Architectures with a direct-mapped cache, such as those considered in this report, are prone to severe performance degradations if the memory access patterns result in cache conflicts. This problem can be addressed by a number of software solutions. Firstly, the memory locations of data can be adjusted so as to attempt to prevent cache conflicts—specifically, I have proposed address randomisation techniques, Iliffe vector shuffling and address optimisation techniques to solve the problem of cache conflicts. Alternatively, the computation itself can be modified, using loop unrolling and scalar variables, so that the memory access patterns do not involve tightly interleaved access to two or more arrays.

Whether these techniques will be of long-term benefit depends upon future developments in computer architecture and compiler technology. At least one new architecture provides a set-associative data cache, eliminating the problem of cache conflicts. One can only hope that this trend continues. In the mean time, researchers and students working with existing systems may find that their applications benefit substantially from the ideas presented in this report.

8 Acknowledgements

Part of this research was performed while the author was visiting in the School of Computer Science, Carnegie Mellon University. This research was supported in part by Digital Equipment Corporation (Australia) Pty Ltd.

References

- [1] John L. Hennessy and David Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.
- [2] Sun Microsystems. Sun systems and their caches. White paper, Sales Tactical Engineering, Sun Microsystems Computer Corporation, 1990.
- [3] Sun Microsystems. The SuperSPARC microprocessor. Technical white paper, Sun Microsystems Computer Corporation, 1992.
- [4] William Stallings. *Computer Organization and Architecture, 2nd ed.*, chapter 14. Macmillan Publishing Company, 1990.

A Program Listing

The program for measuring cache parameters is reproduced below. It, and all the example routines in this report, may be obtained in machine-readable form from the author, by anonymous FTP from `ftp.mpce.mq.edu.au` in the directory `/pub/comp/src/920113.hamey`. The programs are written in C.

```
/* SCCS ID: cache.c 1.1 2/8/93 */

# include <stdio.h>

# define MAXDATA (1<<22)
# define MINSIZE 256
# define MAXSIZE (1<<19)
# define MAXDEPTH 8

int time_read (), time_write (), time_cache ();
int timer_user_ms (), timer_wall_ms ();

main (ac, av)
int ac;
char **av;
{
    int verbose, assoc, size, linesize, i, thistime, prevtime = 0;
    int cache_assoc = 0, cache_size = 0, cache_line = 0;
    int cache_width; /* cache_size divided by cache_assoc */
    int firstline, conflict, nonconflict, timehit, timemiss, niter;
    char *data;
    char *malloc ();

    verbose = ac > 1 && strcmp (av[1], "-v") == 0;

    data = malloc (MAXDATA+MAXSIZE*(MAXDEPTH*2+3));
    if (data == NULL)
    { fprintf (stderr,
        "Memory allocation failed: change MAXDATA and MAXSIZE.\n");
        exit (1);
    }

    /* Ensure that the data array is aligned with the
     * presumed maximum cache size. (It must be aligned
     * with the cache line for proper operation.)
     */
    data = (char *) (( ((long int) (data)) + (MAXSIZE-1) ) &
        ~(MAXSIZE-1));
```

```

/* Initialise. */
for (i = 0; i < MAXDATA+MAXSIZE*(MAXDEPTH*2+2); i++)
    data[i] = i & 0xff;

/* Determine the size of the cache, assuming it is
 * a power of 2 times the associativity */
for (assoc = 1; assoc < MAXDEPTH && cache_size == 0; assoc++)
{
    for (size = 256; size <= MAXSIZE; size <<= 1)
    {
        thistime = time_read (data, size, size, assoc, 1);

        if (verbose)
        {
            printf ("Test for associativity %d, cache size %d: ",
                    assoc, size*assoc);
            timer_display (stdout);
        }

        if (size > 256)
        {
            /* Assume that cache conflict has been detected if
             * performance degrades by 20% or more. */
            if (thistime > prevtime * 6 / 5)
            {
                printf ("Cache associativity is %d.\n", assoc);
                printf ("Cache size is %d bytes\n", size*assoc);
                cache_size = size*assoc;
                cache_assoc = assoc;
                cache_width = size;
                break;
            }
        }
        prevtime = thistime;
    }
}

if (cache_size == 0)
{
    printf ("Cache size estimation failed.\n");
    exit (1);
}

```

```

/* Now determine the cache line size. Start by trying half
 * the cache width. */
firstline = cache_width / 2;
for (linesize = firstline; linesize >= 1; linesize >>= 1)
{
    thistime = time_read (data, cache_width,
        cache_width + (linesize>>1), cache_assoc, 1);

    if (verbose)
    {
        printf ("Test for line size %d: ", linesize>>1);
        timer_display (stdout);
    }

    if (linesize < firstline)
    {
        /* Assume that cache conflict has been detected if
         * performance degrades by 20% or more. */
        if (thistime > prevtime * 6 / 5)
        {
            printf ("Cache line is %d bytes\n", linesize);
            cache_line = linesize;
            break;
        }
    }
    prevtime = thistime;
}

if (cache_line == 0)
{ printf ("Cache line estimation failed.\n");
  exit (1);
}

/* Check whether writes interfere with reads or not.
 * If writes have no effect on reads, then it is write-around.
 * If writes interfere with reads, then it may be
 * fetch on write or it may be simply that the cache read
 * line is invalidated by the write.
 */
nonconflict = time_write (data, cache_width,
    cache_width/2, cache_assoc);
conflict = time_write (data, cache_width,
    cache_width, cache_assoc);

```

```

if (verbose)
    printf ("Write conflict %dms no conflict %dms\n",
            conflict, nonconflict);
if (conflict > nonconflict * 6 / 5)
    printf ("Write misses are not write-around\n");
else
    printf ("Write misses are write-around\n");

for (i = 0; i < 3; i++)
{
    /* Measure the cache miss penalty.
     * Run the read passes, once with gap equal to twice the
     * cache width, so every access is a conflict miss,
     * and once with the gap equal to twice the cache width +
     * twice the cache line, so that the only misses occur 1
     * in every cache line.
     */
    niter = 5 / cache_assoc + 1;
    timehit = time_read (data, cache_width*2, cache_width*2 +
                        cache_line*2, cache_assoc, niter);
    timemiss = time_read (data, cache_width*2, cache_width*2,
                        cache_assoc, niter);
    /* Time difference timemiss - timehit is assumed to be the
     * result of cache missess occurring
     * (MAXDATA - MAXDATA/cache_line) * assoc
     * times per iteration.
     */
    printf ("Cache miss penalty %dns\n",
            (int) (((double) (timemiss - timehit)) * 1000000.0 /
                (cache_assoc + 1) / niter /
                (MAXDATA - MAXDATA / cache_line) + 0.5));
}
exit (0);
}

/* time_read: Basic cache performance measurement routine.
 * The routine performs reads from (assoc+1) locations
 * in each operation. It runs through the data array in
 * a tight loop. The offset between the first two accesses
 * is <gap> and the remaining accesses are separated by
 * <size>. If both <gap> and <size> are the actual cache
 * size divided by the associativity, and <assoc> is at
 * least as large as the actual cache associativity, then
 * cache conflicts will be generated and run time will be

```

```

* large. If <assoc> is less than the actual cache
* associativity then no conflicts will be generated.
* Also, if <gap> differs by at least 1 cache line from
* the cache size divided by the associativity
* then no conflicts will occur.
*/

int time_read (data, size, gap, assoc, niter)
char *data;
int size, gap, assoc, niter;
{
    char *s1, *s2, *s3, *s4, *s5, *s6, *s7, *s8;
    int i, sum;

    s1 = data + gap;
    s2 = s1 + size;
    s3 = s1 + size;
    s4 = s1 + size;
    s5 = s1 + size;
    s6 = s1 + size;
    s7 = s1 + size;
    s8 = s1 + size;
    sum = 0;

    timer_reset ();
    timer_on ();
    while (--niter >= 0)
        switch (assoc)
        {
            case 1:
                for (i = 0; i < MAXDATA; i++)
                {
                    sum += data[i] + s1[i];
                }
                break;
            case 2:
                for (i = 0; i < MAXDATA; i++)
                {
                    sum += data[i] + s1[i] + s2[i];
                }
                break;
        }
}

```

```

case 3:
    for (i = 0; i < MAXDATA; i++)
    {
        sum += data[i] + s1[i] + s2[i] + s3[i];
    }
    break;
case 4:
    for (i = 0; i < MAXDATA; i++)
    {
        sum += data[i] + s1[i] + s2[i] + s3[i] + s4[i];
    }
    break;
case 5:
    for (i = 0; i < MAXDATA; i++)
    {
        sum += data[i] + s1[i] + s2[i] + s3[i] + s4[i] +
            s5[i];
    }
    break;
case 6:
    for (i = 0; i < MAXDATA; i++)
    {
        sum += data[i] + s1[i] + s2[i] + s3[i] + s4[i] +
            s5[i] + s6[i];
    }
    break;
case 7:
    for (i = 0; i < MAXDATA; i++)
    {
        sum += data[i] + s1[i] + s2[i] + s3[i] + s4[i] +
            s5[i] + s6[i] + s7[i];
    }
    break;
case 8:
    for (i = 0; i < MAXDATA; i++)
    {
        sum += data[i] + s1[i] + s2[i] + s3[i] + s4[i] +
            s5[i] + s6[i] + s7[i] + s8[i];
    }
    break;
}
timer_off ();

```

```

    /* Assign the result somewhere to prevent a really
    * good optimiser from removing the entire loop. */
    data[0] = sum;
    return timer_user_ms ();
}

/* time_write: Used to determine whether writes interfere with
* cached read data from conflicting locations.
* The routine iteratively performs a computation involving
* writing to data[i] while reading from data[i+gap],
* data[i+gap+width], data[i+gap+width*2], ...
* If gap and width are both the cache width (size divided by
* associativity), then the write is at a potential conflict
* location. If gap is (say) halved, then the write will not
* conflict. If there is a timing difference, it indicates that
* write operations interfere with cached read data.
*/

int time_write (data, width, gap, assoc)
char *data;
int width, gap, assoc;
{
    char *s1, *s2, *s3, *s4, *s5, *s6, *s7, *s8;
    int i, sum;

    s1 = data + gap;
    s2 = s1 + width;
    s3 = s1 + width;
    s4 = s1 + width;
    s5 = s1 + width;
    s6 = s1 + width;
    s7 = s1 + width;
    s8 = s1 + width;

    timer_reset ();
    timer_on ();
    sum = 0;
    switch (assoc)
    {
        case 1:
            for (i = 0; i < MAXDATA; i++)
                data[i] = s1[i];
            break;
    }
}

```

```

case 2:
    for (i = 0; i < MAXDATA; i++)
        data[i] = s1[i] + s2[i];
    break;
case 3:
    for (i = 0; i < MAXDATA; i++)
        data[i] = s1[i] + s2[i] + s3[i];
    break;
case 4:
    for (i = 0; i < MAXDATA; i++)
        data[i] = s1[i] + s2[i] + s3[i] + s4[i];
    break;
case 5:
    for (i = 0; i < MAXDATA; i++)
        data[i] = s1[i] + s2[i] + s3[i] + s4[i] + s5[i];
    break;
case 6:
    for (i = 0; i < MAXDATA; i++)
        data[i] = s1[i] + s2[i] + s3[i] + s4[i] + s5[i] +
            s6[i];
    break;
case 7:
    for (i = 0; i < MAXDATA; i++)
        data[i] = s1[i] + s2[i] + s3[i] + s4[i] + s5[i] +
            s6[i] + s7[i];
    break;
case 8:
    for (i = 0; i < MAXDATA; i++)
        data[i] = s1[i] + s2[i] + s3[i] + s4[i] + s5[i] +
            s6[i] + s7[i] + s8[i];
    break;
}
timer_off ();
data[0] = sum;
return timer_user_ms ();
}

```

```

/* Routines to implement a simple timer mechanism. */

# include <sys/time.h>
# include <sys/timeb.h>
# include <sys/resource.h>

/* Static data structures used by abstract object timer. */

static struct timeb wall_on, wall_off;
static struct rusage user_on, user_off;
static long int wall, user;

timer_on ()
{ ftime (&wall_on);
  getrusage (RUSAGE_SELF, &user_on);
}

timer_off ()
{ getrusage (RUSAGE_SELF, &user_off);
  ftime (&wall_off);
  wall += (wall_off.time - wall_on.time) * 1000 +
    wall_off.millitm - wall_on.millitm;
  user += (user_off.ru_utime.tv_sec - user_on.ru_utime.tv_sec) *
    1000 + (user_off.ru_utime.tv_usec -
    user_on.ru_utime.tv_usec) / 1000;
}

timer_display (out)
FILE *out;
{ fprintf (out, "wall %d.%03d user %d.%03d\n",
  wall/1000, wall % 1000, user / 1000, user % 1000);
}

int timer_user_ms ()
{ return user;
}

int timer_wall_ms ()
{ return wall;
}

timer_reset ()
{ wall = user = 0;
}

```

About the Author



Leonard G. C. Hamey is a Lecturer in Computing at Macquarie University. His research interests centre on computer vision, and include low-level processing, high-level perception and the application of the connectionist and perceptual organisation paradigms to this area. Dr. Hamey received his BSc(hons) in Statistics from Macquarie University in 1982 and his PhD in Computer Science from Carnegie Mellon University in 1988. He is a member of IEEE and the Australian Pattern Recognition Society. Besides his professional involvements, Leonard is a Christian and active in his local church.

“The scientific picture of the world around me is very deficient. It gives a lot of factual information, puts all our experience in a magnificently consistent order, but it is ghastly silent about all . . . that is really near to our hearts, that really matters to us. . . . it knows nothing of beautiful and ugly, good or bad, God and eternity. Science sometimes pretends to answer questions in these domains, but the answers are very often so silly that we are not inclined to take them seriously. . . . Whence came I and whither go I? That is the great unfathomable question, the same for every one of us. Science has no answer to it.”

— *Erwin Schroedinger, Nobel laureate.*