

# D 1 Optimization of Numerical Codes

S. Goedecker  
Institut für Physik  
Universität Basel

The material of this article is taken with permission of SIAM from the book: S. Goedecker, A. Hoisie: "Performance Optimization of Numerically Intensive Codes", SIAM publishing company, Philadelphia, USA 2001 (ISBN 0-89871-484-2)

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Notions of computer architecture</b>	<b>3</b>
2.1	The on-chip parallelism of superscalar architectures . . . . .	3
2.2	Overview of the memory hierarchy of modern architectures . . . . .	5
2.3	Mapping rules for caches . . . . .	6
2.4	A taxonomy of cache misses . . . . .	7
2.5	TLB misses . . . . .	8
2.6	Multi-level cache configurations . . . . .	8
<b>3</b>	<b>A few basic efficiency guidelines</b>	<b>8</b>
3.1	Selection of best algorithm . . . . .	9
3.2	Use of efficient libraries . . . . .	9
3.3	Optimal data layout . . . . .	10
3.4	Use of Compiler optimizations . . . . .	10
<b>4</b>	<b>Timing and profiling of a program</b>	<b>11</b>
4.1	Subroutine level profiling . . . . .	11
4.2	Timing small sections of your program . . . . .	12

---

<b>5</b>	<b>Optimization of floating point operations</b>	<b>13</b>
5.1	Fused Multiply-Add instructions . . . . .	14
5.2	Exposing Instruction Level Parallelism in a Program . . . . .	14
5.3	Improving the ratio of floating point operations to memory accesses . . . . .	15
5.4	Aliasing . . . . .	16
5.5	Special functions . . . . .	20
5.6	If statements . . . . .	20
5.7	Loop overheads . . . . .	21
5.8	Copy overheads in Fortran90 . . . . .	21
<b>6</b>	<b>Optimization of memory access</b>	<b>21</b>
6.1	Loop reordering for optimal data locality . . . . .	21
6.2	Cache thrashing . . . . .	23
6.3	Square blocking . . . . .	25
6.4	Line blocking . . . . .	27
6.5	Prefetching . . . . .	31

# 1 Introduction

Modern computers are highly complex machines which typically deliver only a very small fraction of the theoretically possible peak speed for ordinary programs. Better performance can be achieved if the architectural features are already exploited during the development of a program. The basic knowledge for writing efficient code will be presented.

## 2 Notions of computer architecture

The computer architectures that are at the focus of this article are “superscalar” type architectures, characterized by on-chip instruction parallelism. All the workstations of major vendors belong to this category. Superscalar only refers to the execution of instructions and does not constrain other architectural features. We will analyze in this article superscalar microprocessors with a hierarchical memory structure. Because of the memory hierarchy, the memory access times are not independent of the location of the data in memory.

### 2.1 The on-chip parallelism of superscalar architectures

Superscalar architectures can perform several operations in one cycle. An  $n$ -way superscalar processor is capable of fetching up to  $n$  instructions per cycle. Typically, a processor can perform a combination of adds, multiplies, loads/stores and branching instructions in one cycle. This instruction-level parallelism can be implemented in hardware or by a combination of hardware and software support. In hardware, a processor can have several units that work concurrently. Instruction-level parallelism can be increased by special instructions, such as the fused multiply-add (section 5.1), that allows the execution of a floating point multiplication and a consecutive addition in one assembler instruction.

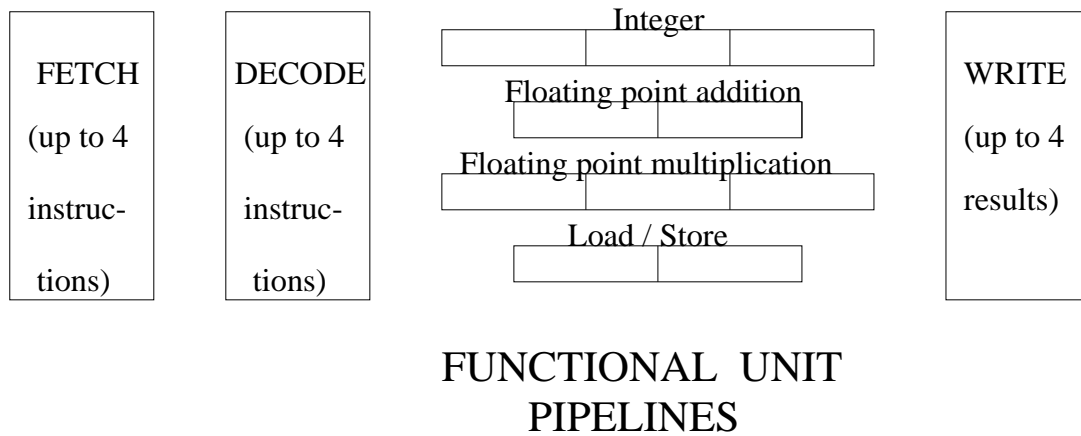
All modern processors are pipelined. Processor pipelines take advantage of the fact that all instructions can be processed in several smaller subtasks, each of which can be done within one clock cycle. The pipeline is typically subdivided into the following stages:

- Fetch instruction: instructions are fetched from the instruction cache into the processor
- Instruction decode
- Issue and execute: instructions are issued to different functional units and are executed. Some functional units, such as the floating points unit, are often again pipelined. We call this “functional unit pipelining” to differentiate it from processor pipelining.
- Write back and commit: instructions broadcast the execution result to other instructions and commit the results to the registers.

A schematic diagram of a processor of the type described in the preceding text is shown in Figure 1.

The functional unit pipeline requires a more explicit description. This pipelining means that, besides the ability of executing different instructions in different functional units, instructions can be executed in the same functional unit simultaneously at different pipeline stages. Frequently two multiplications and two additions can be processed simultaneously in each of these units when the pipeline is full, each instruction being processed in one of the three stages of the

# PROCESSOR PIPELINE



**Fig. 1:** Schematic diagram of a hypothetical four-way superscalar pipelined processor with two floating point pipelines, one integer pipeline and one load/store pipeline. The depth of these functional unit pipelines varies between two and three cycles in this processor. Consequently, the depth of the entire processor pipeline varies between five and six cycles.

pipeline. We say in this case that the pipeline has a depth of three. In each cycle, work on a new instruction can start in the pipeline, but it takes three cycles for its completion.

The latency of dependent operations is given by the depth of the functional unit pipeline and not by the depth of the whole processor pipeline. For a single stream of dependent operations, modern processors can typically finish a result every two to four cycles.

Special instructions, such as the fused multiply-add already mentioned under the aspect of increased parallelism, are also a means to reducing the operation's latency.

Processors with out-of-order execution capabilities show less performance degradation in the presence of dependencies, if several independent streams of dependent instructions can be processed. On such architectures, a queue of several instructions waiting to be executed is maintained. Instructions for which all the operands are available are executed. The processor is dynamically switching back and forth between different streams, working only on those for which the pipeline will not stall because of missing operands.

The depth of the whole processor pipeline becomes visible as the latency of a mispredicted branch. This typically happens at the end of a loop. The program flow has to jump to a new location and start feeding the processor pipeline with a new stream of instructions. Branch prediction, found on some processors, helps to reduce the number of mispredicted branches by extrapolating the program flow pattern from data accumulated during the run.

A pipeline that cannot accept a new instruction at a certain stage is called "stalled". Several reasons for such stalls are possible. Two of the reasons, dependent operations and branches were mentioned above. Other causes for stalls are due to memory access. Either the consecutive instruction is still on its way from the memory towards the processor, or, more likely, a numerical operand that has to be loaded is not yet available due to a cache miss. Keeping all the pipelines busy without stalling is one of the key ingredients for achieving high performance

on superscalar processors. From the point of view of the programmer, a pipeline with a latency of  $s$  cycles has practically the same effect as  $s$  separate floating point units that have a latency of just one cycle. This assertion will become clear in the discussion of the subroutine “lngth4” in section 5.2.

## 2.2 Overview of the memory hierarchy of modern architectures

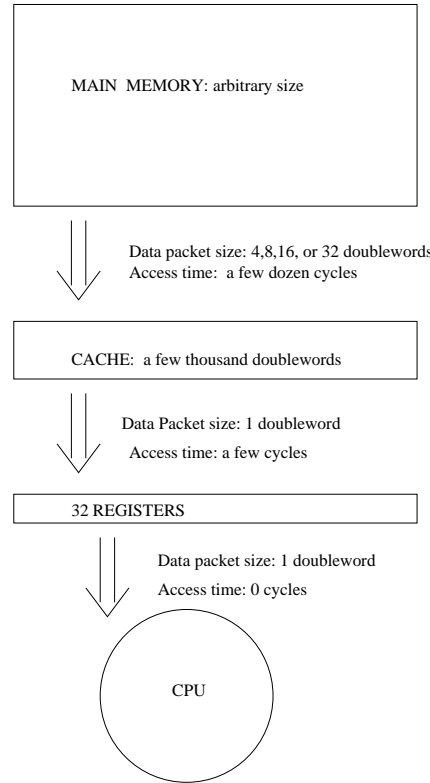
The memory subsystem of computers is organized hierarchically. It can be visualized as a layered inverted pyramid. Layers at the top are big, but have slow access times. Layers towards the bottom of the pyramid are small but have fast access times since they are close to the CPU. The size of the data packets transferred among the different layers varies as well. Data movement between the layers at the top involves larger amounts of data. The chunks of data transferred decreases to as little as one word at the bottom of the memory hierarchy. We will now describe this reversed pyramid in more detail, starting at the bottom with the smallest layer.

The fastest, but smallest memory level is composed of the registers, placed on the chip, housing the Central Processing Unit (CPU). IBM processors have 32 floating point logical registers that can be accessed without any delay (0 cycle latency). Pentiums and opterons have fewer registers. In 64 bit mode they have 16 general purpose registers and 16 SSE vector registers. In 32 bit mode these numbers are cut into half. In addition there are integer registers. They are used for integer arithmetic, including the calculation of the addresses of array elements that are loaded or stored. In this context it is worthwhile pointing out that the number of physical registers can be larger than the number of logical registers.

The next level in the memory hierarchy is the cache. Many machines have multiple cache levels, a level-1 (L1) cache and a level-2 (L2) cache, for example. An L1 cache can usually be accessed with a 1-cycle latency, if the data is available in the cache. Since in many cases a load or store from the L1 cache to a register can be overlapped with floating point operations, this 1-cycle latency can often be hidden, giving the appearance of a 0-cycle access time. Data transfer from the L1 cache to the registers takes place in units of words (i.e. single or double precision numbers). If the data is not in the L1 level, it has to be fetched from the higher level of the memory hierarchy, the memory or the L2 cache. A “cache miss” occurs and the program execution has to wait for several cycles until the data are transferred into the L1 cache. In the same way, an L2 cache miss can occur, necessitating a data transfer from the main memory. The penalty for a L2 cache miss is larger than for a L1 cache miss. The penalty is typically in the range of a few dozen cycles for an L1 miss, and as high as 100 cycles for an L2 miss. The smallest possible unit that can be loaded from memory into the cache is a cache line, typically comprising between 4 and 32 words. The cache line sizes for the various cache levels can be different. A typical memory hierarchy is shown in Figure 2.

At this point it is important to note that although we will be concentrating in this article on data caches, L1 instruction caches are also present. The goals of the instruction cache are very similar to those of the data cache, only applied to instructions. Frequently used instructions are accommodated closer to the CPU in order to achieve faster access times. If two cache levels exist, then typically the second level cache is used for both data and instructions.

The next generic hierarchy level is the memory attached to one processor. The slow access time (high latency) of the CPU to data in the main memory is a major bottleneck in many scientific applications. Another, equally important bottleneck, is the limited memory bandwidth, i.e. the limited amount of data that can be transferred from the memory towards the lower levels of the memory hierarchy. One architectural technique leading to an increased memory bandwidth is



**Fig. 2:** Schematic view of a memory hierarchy

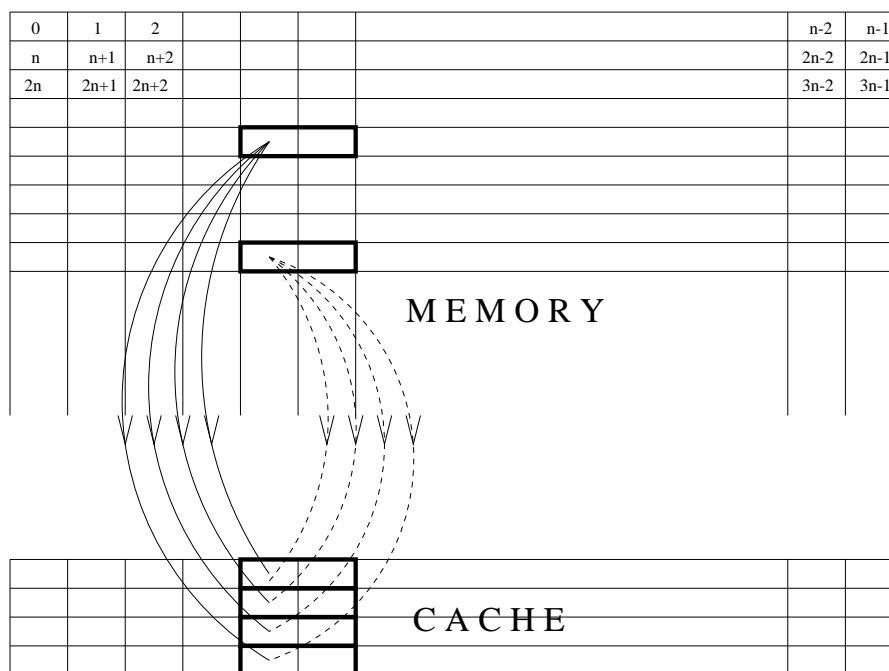
memory interleaving. Some workstation have four memory banks. If a cache line, 32 words in size, is brought into cache from memory, then the first eight words are filled from the first memory bank, the second eight words form the second bank and so on. The memory bandwidth is increased four times compared to the case of a single memory bank. However, memory interleaving has no beneficial effect on the latency. If only a single word of the 32 words of a cache line is needed, the memory access time is latency bound and would not benefit from memory interleaving. Data access with stride one is therefore faster compared with larger stride access.

## 2.3 Mapping rules for caches

In a fully associative cache, a cache line can be placed in any free slot in the cache. Fully associative caches are rarely used for data caches, but are sometimes for TLB (see 2.5). In the standard non-fully associative cache, a cache line that is loaded from the main memory can only be placed in a limited number of locations in the cache. Based on the number of such possible locations, we distinguish between directly and indirectly mapped caches. In a directly mapped cache, there is only one possible location. Indirectly mapped caches allow for more than one location.

If the size of the cache is  $n$  words, then the  $i$ th word in memory can be stored only in the position given by  $\text{mod}(i, n)$  in directly mapped caches. If we have an  $m_{as}$ -way associative cache of size  $m_{as} \times n$ , then any location in memory can be mapped to  $m_{as}$  possible locations in cache, given by the formula  $j \times n + \text{mod}(i, n)$ , where  $j = 0, \dots, m_{as} - 1$ . The situation for a 4-way associative cache with a cache line size of two words is illustrated in Figure 3.

If, in an  $m_{as}$ -way associative cache, all the  $m_{as}$  possible locations for a data set are taken, one



**Fig. 3:** Mapping rules for a 4-way associative cache. Each memory location can be mapped to four cache locations, but many memory locations map to the same four cache locations. Since we assumed that the cache line size is two words, the portion to be transferred is the framed area.

of them has to be overwritten.

## 2.4 A taxonomy of cache misses

Cache misses can be categorized as follows:

- **Compulsory cache misses:** These misses occur when the cache line has to be brought into the cache when first accessing it. They are unavoidable.
- **Capacity cache misses:** They are related to the limited size of the cache preventing all the necessary data to be simultaneously in the cache. New data brought into the cache may have to overwrite older entries.
- **Conflict cache misses:** These misses occur in directly mapped or set associative caches. Because of the mapping rules, the effective cache size is usually smaller than the physical cache size. The effective cache size would be equal to the physical cache size only if a data item from the main memory could go into any location in the cache. Since this is not the case, unoccupied cache lines slots will frequently be found in the cache, thus reducing the effective cache size. The extreme case, when most of the physical cache space is not available because of these mapping rules, is called cache thrashing.

Programming techniques leading to a reduction of the number of cache misses will be discussed later.

## 2.5 TLB misses

Modern workstations are virtual memory machines. This means that they distinguish between a logical and a physical memory address. The logical address is the one utilized in a program to identify array elements. The array elements  $x(100)$  and  $x(600)$  have the logical address of 100 and 600 respectively. However, they are not necessarily at the physical locations 100 and 600 in the memory. Logical addresses are translated into physical ones as described below.

Each address belongs to a page in the memory. The size of a memory page varies from machine to machine. For example, on the IBM 590 one page holds  $2^9 = 512$  doublewords (one doubleword = 8 bytes). In this case, the array element  $x(100)$  would be on the first logical page, while  $x(600)$  would belong to the second logical page. A page can be stored in any physical page slot in memory or even on disk.

A page table, stored in the main memory, keeps track of the mapping from physical to logical addresses. The most frequently used entries of that table are stored in a special cache called the Translation Look-Aside Buffer (TLB). Thus, when accessing a logical memory location that belongs to a logical page whose mapping to a physical address is not present in the TLB, that page location has to be fetched from the much larger page table. A TLB miss will occur, very similar to a cache miss for all practical purposes, only costlier.

From the point of view of the programmer, the effect of a TLB is exactly the same as if there was an extra cache level, whose cache line size is equal to the page size and whose overall size is equal to the total number of words contained in all the pages whose physical addresses can be held in the TLB. For optimal performance, any data item has to be contained in both memory hierarchy levels. In other words both the capacity misses and the conflict misses have to be minimized not only for the real cache levels but also for the TLB level. On an IBM 590, the TLB is a two-way associative cache with  $2 \times 256$  entries each holding 512 doublewords. Its effect is therefore the same as an additional two-way associative cache level with a total capacity of  $2^{18}$  words and a cache line length of 512. The TLB and the data caches are accessed in parallel. Thus, in the event of a combined cache and a TLB miss, the latency is smaller than the sum of the individual latencies.

## 2.6 Multi-level cache configurations

Most of the modern computers have two or even three cache levels. The reason for this lies in the tradeoffs that a computer architect has to make in order to achieve a balanced, high-performance architecture. An L1 cache is usually built on-chip. Given the limitations on the number of transistors on a chip and the high cost of a cache in terms of transistor consumption, a tradeoff is made between the size of the cache and the features and complexity of the CPU. For these on-chip “real-estate” reasons, the larger L2 caches are built off-chip. Because of the longer signal traveling time, off-chip caches are slower in access time, but they can be significantly larger than the L1 caches.

## 3 A few basic efficiency guidelines

In this chapter we will not dive deeply into sophisticated optimization techniques. Rather we will propose some basic efficiency guidelines that can improve program performance without any significant time investment by the programmer.



### 3.1 Selection of best algorithm

Usually several algorithms are available for a specific problem. Choosing the algorithm that is best suited for the problem of interest and the target computer clearly is the first and most important step. Frequently, the complexity or scaling behavior of two algorithms is different. For instance, the number of operations in a matrix-matrix multiplication is  $N^3$  for  $N \times N$  matrices with the ordinary algorithm, but only  $N^{2.8}$  with the Strassen algorithm. For large data sets, the advantage of using a low complexity algorithm (the Strassen algorithm in the matrix multiplication example) is overwhelming and certainly dominating other suitability aspects. If, under such circumstances, the implementation of a low complexity algorithm is not well adapted to the computer architecture, the advantages of using it could be lost. The common sense solution is to find a better implementation of this algorithm instead of using a potentially high-performance implementation of the higher complexity algorithm. In a different situation, if two algorithms differ in their operation count only by a small factor, the potential for a good implementation on a specific architecture may be the dominating aspect. This article will present the information necessary to judge the suitability of different algorithms for various architectures.

### 3.2 Use of efficient libraries

For most of the basic computational problems high quality numerical libraries are available today. Instead of duplicating work by developing and optimizing one's own version, one should definitely use such library routines whenever available.

For most basic dense linear algebra computations, the BLAS (Basic Linear Algebra Subroutines) (<http://www.netlib.org/index.html>) is a highly recommendable library, particularly when provided in optimized form by computer vendors for their specific architecture. Since the calling sequences is the same in all vendor-optimized implementations, a program using BLAS routines is portable to any computer.

There are three levels of BLAS. Level-1 contains vector-vector kernels, such as a scalar product between two vectors, level-2 deals with matrix-vector operations, such as matrix times vector multiplication and level-3 contains matrix-matrix routines, such as matrix times matrix multiplication. For reasons that will be discussed in section 5.3, the higher BLAS level routines run faster than the lower level ones. As shown in Table 1, optimized Level-3 BLAS frequently come close to the peak speed of the machine. By reordering loops in a code it is frequently possible to replace several calls to lower level BLAS by a single call to a higher level routine. Even though the vendor supplied BLAS routines are usually well optimized, exceptions exist,

**Table 1:** Comparison of the speed of different levels of BLAS routines on an Intel Xeon running at 2.4 GHz.

BLAS routine	Speed (Mflops)
DGEMM (Level-3: matrix matrix mult.)	3500
DGEMV (Level-2: matrix vector mult.)	550
DDOT (Level-1: scalar product)	500

particularly on new computer models. If the time spent in BLAS routines is significant, it is a good practice to verify by timing (as it will be explained in section 4) whether the performance of BLAS is reasonable. If the loops are very short, the overhead of calling a BLAS routine can

become prohibitive. Even a moderately optimized user-written loop structure can be faster in this case. Cray compilers automatically replace user-written loop structures by matching level-2 and level-3 BLAS routines, unless the loops are expected to be short.

A very good quality public domain library is LAPACK, containing all the standard dense linear algebra operations, such as the solution of linear systems of equations, singular value decompositions and eigenvalue problems. It supersedes the older LINPACK and EISPACK libraries. This library is built on top of the BLAS library leading to very good performance, particularly if a vendor optimized version of BLAS is utilized. Some of the LAPACK routines can also be found in optimized form in some of the vendors' mathematical libraries. Further information on LAPACK can be found in its users' guide at <http://www.netlib.org/index.html>.

In addition to these public domain libraries, several computer manufacturers provide scientific libraries containing additional routines, such as special functions, Fast Fourier Transforms, integration and curve fitting routines.

### 3.3 Optimal data layout

This very important aspect of any optimization work, the layout of all the data structures needed in the calculation, should take place before the writing of the program starts. Data that are processed simultaneously should be located close to each other in the physical memory. This will ensure "data locality", meaning that data that are brought in cache will be used at least once before being flushed out of cache. Thus, the high cost of a cache miss is distributed among several memory accesses.

For instance, a Fortran array containing the positions of a collection of  $n$  particles should be dimensioned as

```
dimension r(3,n)
```

instead of

```
dimension r(n,3)
```

since most likely the three spatial coordinates for a given particle will be accessed consecutively.

### 3.4 Use of Compiler optimizations

Programs compiled without any optimization usually run very slowly. Using a medium optimization level (-O2 on many machines) typically leads to a speedup by factors of two to three without a significant increase in compilation time. Using the highest available optimization levels can lead to further performance improvements, but to performance deterioration as well. The compiler applies transformation rules, based on heuristics, that in most cases improve performance. Since the compiler does not have all the necessary information to determine whether certain transformations of the program will pay off, the success of the transformations is not guaranteed. It is certainly worthwhile to try several optimization levels and possibly some other compiler options too and assess their effect on the overall program speed.

## 4 Timing and profiling of a program

The starting point of any optimization work is the timing and profiling of the program. Timing and profiling are the means to determine the performance of a code. The measured time, together with either estimates or counts of the number of floating point operations, allows us to calculate the speed usually measured in Mflops (Millions of Floating Point Operations per Second). A comparison between the measured Mflops and the peak Mflops rates for that machine gives a good indication of the efficiency of your program. The dividing line between acceptable and poor performance is, of course, disputable. One rule-of-thumb is that for large-scale scientific applications 50 % of peak performance is very good, albeit hard to achieve, whereas less than 10 % should be an indication that optimization work is in order. In this order of magnitude type of analysis, contrasting the performance of your code against existing benchmarks performing similar computations can be helpful, too.

For the interpretation of the output of the timing and profiling tools it is necessary to understand the various time metrics that are used.

The most important time measure is the CPU time. This is the time in which the CPU is dedicated to the execution of a program. It is the sum of two parts, the user time and the system time. The user time is the time spent executing the instructions of the program. The system time is the time spent by the operating system for service operations requested by the program. In general, the system component of the CPU time is small compared to the user component of the CPU time. If it is not, this is an indication of an inefficiency in the code.

Another important metric is the elapsed or wallclock time. If several programs are running on a computer, the total execution time of one program will be larger than its CPU time. Assuming that three programs are running with the same priority, the elapsed time will be roughly three times the CPU time. Even in standalone (i.e., when a single code is running on the computer), the CPU will timeshare between that code and pure systems tasks. Because of this, the elapsed time will be larger than the CPU time.

Overall timing information of a program can be obtained using the UNIX commands “timex” or “time”. This simply involves preceding the running command for the code, say “a.out”, by the “timex” or “time” command, such as in “time a.out”. At the end of the run, the CPU time, split up in user and system time, and the elapsed time will be printed out. The exact format of this timing output varies among different vendors, but is described in the man pages.

Because of the ambiguities and interference effects mentioned above, there are always considerable fluctuations in measured times. Even in standalone, fluctuations of a few percentage points are normal. When the machine is shared with other jobs, these fluctuations can be even bigger. The first thing to know when starting to optimize a program is which parts of it take most of the runtime, i.e. the identification of the “hotspots”. Profiling the program will answer this question. Several profiling techniques are available and will be discussed in the next sections.

### 4.1 Subroutine level profiling

A subroutine level profile is obtained by compiling the program with a special compiler flag, usually “-p”. The compiler will then insert timing calls at the beginning and end of each subroutine. At execution time, the timing information is written to a file, typically “mon.out”. The “prof” command allows you to read in the information contained in this file and to analyze the code in the form of a “profile”. Let us profile the following short program:

```
implicit real*8 (a-h,o-z)
```

```

        parameter (nexp=13, n=2**nexp)
        dimension x(n)
        do 15, i=1, n
15      x(i)=1.234d0
        do ic=1, 1000
            call sub1(n, x, sum1)
            call sub2(n, x, sum2)
        enddo
        write(6, *) sum1, sum2
    end

    subroutine sub1(n, x, sum)
    implicit real*8 (a-h, o-z)
    dimension x(n)
    sum=0.d0
    do 10, i=1, n
10      sum=sum+2.d0*x(i) + (x(i)-1.d0)**2+3.d0+i*1.d-20-4.d0*(i-200)**2
    return
    end

    subroutine sub2(n, x, sum)
    implicit real*8 (a-h, o-z)
    dimension x(n)
    sum=0.d0
    do 10, i=1, n
10      sum=sum+x(i)**.3333333333d0
    return
    end

```

On an IBM Power2, the output obtained from the “prof” command looks like this:

Name	%Time	Seconds	Cumsecs	#Calls	msec/call
._pow	63.8	21.11	21.11	8192000	0.0026
.__mcount	28.4	9.39	30.50		
.sub1	3.8	1.27	31.77	1000	1.270
.sub2	2.3	0.77	32.54	1000	0.770
.					
.					

First, it is apparent that the time spent in the compiler library functions is not attributed to the subroutines from which they are called, but to the functions themselves (the exponentiation function is called from subroutine sub2). As a result, if the same library function is called from several subroutines, it is not possible to find out directly which subroutines invokes them more frequently and thus takes more time. Second, we note an additional category “mcount”, which is the time spent in the timing routines themselves, i.e. the overhead of profiling. This can be a significant fraction of the total execution time and it can bias the timing analysis. Calls to subroutines that are part of libraries, such as BLAS, cannot be analyzed in this way.

## 4.2 Timing small sections of your program

Very frequently one wants to know the CPU time spent in a code’s hotspots. By calculating the number of floating point operations executed in hotspots, the timing information allows the calculation of the sustained speed. This is done by manually inserting timing calls.

```
real t1,t2
integer count1,count2,count_rate,countmax

call system_clock(count1,count_rate,count_max)
call cpu_time(t1)

.. code to be timed ...

call cpu_time(t2)
call system_clock(count2,count_rate,count_max)

print*, 'CPU time (sec)',t2-t1
print*, 'elapsed time (sec)',(count2-count1)/float(count_rate)
```

The Fortran90 standard provides for two functions that measure the CPU time and the elapsed time, as shown above. As in the case of profiling, the time spent in these timing routines, i.e. the overhead of timing, can be significant if they are called very frequently.

A possible problem with this analysis method is that the resolution of the timing routines is usually fairly coarse (1/100 of a second for 'mclock', 1/1000 for most others). In order to get good statistics, the runtime of the hotspot has to be much longer than this resolution. Frequently this necessitates artificially repeating the execution of the hotspot by bracketing it with an additional timing loop. Since in these repeated executions data can be available in cache at the beginning of a new timing iteration, the performance numbers can be artificially high. In order to avoid this pitfall, a call to a cache flushing routine has to be inserted at the beginning of each timing iteration. Any routine that refreshes all the cache lines can be used as a cache flushing routine. In timing runs one has to make sure that no floating point exceptions (such as overflows) are present, since they will increase the CPU time. Testing with zeroes can be problematic too on some machines. Some compilers are able to figure out that the numerical results of the timing loops are never utilized and do not execute those loops at all. In this way, the speed of the program can get close to infinity! Therefore, a good practice is to print out some of the numerical results of a timing run in order to make sure that the compiler is not outsmarting you.

## 5 Optimization of floating point operations

As we have pointed out several times, memory access problems are usually the single most detrimental factor leading to large performance degradation. Nevertheless, we will start our optimization discussion with the topic of floating point operations. As a matter of fact, when tuning a program, it is recommendable to start with addressing this type of optimizations before moving on to memory access optimizations. In order to discern the effect of floating point optimizations, it is necessary to eliminate limiting effects due to memory access issues. In the floating point optimization phase this can be easily achieved by working with small data sets that fit in cache, even though they may not represent the memory requirements of a realistic application. Consequently, we will assume in this section that all the necessary data are available in cache and that therefore cache misses do not limit performance. The topic of memory access optimization will be taken up in detail in the next section.

In general, floating point operations dominate in “number-crunching” scientific codes. This is in contrast to other applications, such as compilers or editors, that are integer arithmetic bound. We will not analyze in detail integer performance by itself in this article. Even though the “useful”

operations in most of the scientific codes are floating point operations, integer operations occur frequently too. For example, they are needed to calculate the addresses of array elements, with the effect that practically all load or store operations require integer arithmetic. As a matter of fact, the load and store operations on several architectures are handled by the integer unit. In this article we will not distinguish between the calculation of the address and the process of transferring data from the cache into the registers, but instead consider these two steps as a single operation.

## 5.1 Fused Multiply-Add instructions

The Fused Multiply-Add instruction (FMA), found on several modern architectures, provides for the execution of a multiplication followed by an addition ( $a+b*c$ ) as a single instruction. This reduces the latency compared to the case where a separate multiplication and addition is used and makes the instruction scheduling easier for the compiler.

## 5.2 Exposing Instruction Level Parallelism in a Program

As mentioned in section 2.1, superscalar CPUs have a high degree of on-chip parallelism. The same degree of parallelism has to be exposed in the program in order to achieve the best efficiency. As an illustration, let us look at a simple vector norm calculation.

```

program length
parameter(n=2**14)
dimension a(n)
subroutine lngth1(n,a,tt)
implicit real*8 (a-h,o-z)
dimension a(n)
tt=0.d0
do 100, j=1,n
    tt=tt+a(j)*a(j)
100 continue
return
end

```

We aim to optimize this vector norm subroutine on an IBM 590 workstation. This processor has two floating point units. However, the program has only one independent stream and thus cannot keep two units busy. Furthermore, we see that the output *tt* of one fused multiply-add is the input for the next fused multiply-add. Hence, we have a dependency. Since the latency of the FMA is two cycles, we cannot start a new FMA every cycle, as it would be possible if we had independent operations. With this in mind we restructure the program as follows.

```

subroutine lngth4(n,a,tt)
c works correctly only if the array size is a multiple of 4
implicit real*8 (a-h,o-z)
dimension a(n)
t1=0.d0
t2=0.d0
t3=0.d0
t4=0.d0
do 100, j=1,n-3,4
c first floating point unit, all even cycles

```

```

        t1=t1+a(j+0)*a(j+0)
c first floating point unit, all odd cycles
        t2=t2+a(j+1)*a(j+1)
c second floating point unit, all even cycles
        t3=t3+a(j+2)*a(j+2)
c second floating point unit, all odd cycles
        t4=t4+a(j+3)*a(j+3)
100      continue
        tt=t1+t2+t3+t4
        return
end

```

This kind of transformation is called loop unrolling. Its effect is that it usually improves the availability of parallelism within a loop. The number of independent streams generated by loop unrolling, four in the subroutine `lngh4`, is called the depth of the unrolling. On an IBM 590 workstation, the unrolled version runs at peak speed (260 Mflops), whereas the original version runs at just one fourth of the peak speed. Of course, this is true only for data sets that are cache resident.

Most compilers will try to do optimizations of this type when invoked with the highest optimization level. Even in this simple example some subtleties are present and it can not be taken for granted that the compilers generate optimal code.

### 5.3 Improving the ratio of floating point operations to memory accesses

In addition to the advantages already discussed, loop unrolling can also be used to improve the ratio of floating point operations to load/stores. This improved ratio can be beneficial for loops that have many array references. Such loops are dominated by memory accesses. Let us look at the following matrix-vector multiplication routine, multiplying the vector by the transposed matrix.

```

subroutine mult(n1,nd1,n2,nd2,y,a,x)
implicit real*8 (a-h,o-z)
dimension a(nd1,nd2),y(nd2),x(nd1)
do i=1,n2
  t=0.d0
  do j=1,n1
    t=t+a(j,i)*x(j)
  enddo
  y(i)=t
enddo
return
end

```

First of all, we note that the loop ordering is optimal from the point of view of data locality. All the memory accesses have unit stride. If the loop over  $i$  were the innermost one, the stride would be  $nd1$  for accesses to  $a$ . The problem with this subroutine is that, even though the vector  $x$  has only  $n1$  elements, we are loading  $n1 \times n2$  elements of  $x$ . Expressed differently, each element of  $x$  is loaded  $n2$  times. This is clearly a waste of memory bandwidth.

The subroutine “multo” shown below is an unrolled version of “mult”. The effect to be demonstrated here is related to a better ratio of floating point to load/stores in the optimized version “multo”.

```

      subroutine multo(n1,nd1,n2,nd2,y,a,x)
c works correctly only if n1,n2 are multiples of 4
      implicit real*8 (a-h,o-z)
      dimension a(nd1,nd2),y(nd2),x(nd1)
      do i=1,n2-3,4
        t1=0.d0
        t2=0.d0
        t3=0.d0
        t4=0.d0
        do j=1,n1-3,4
          t1=t1+a(j+0,i+0)*x(j+0)+a(j+1,i+0)*x(j+1)+a(j+2,i+0)*x(j+2)+a(j+3,i+0)*x(j+3)
          t2=t2+a(j+0,i+1)*x(j+0)+a(j+1,i+1)*x(j+1)+a(j+2,i+1)*x(j+2)+a(j+3,i+1)*x(j+3)
          t3=t3+a(j+0,i+2)*x(j+0)+a(j+1,i+2)*x(j+1)+a(j+2,i+2)*x(j+2)+a(j+3,i+2)*x(j+3)
          t4=t4+a(j+0,i+3)*x(j+0)+a(j+1,i+3)*x(j+1)+a(j+2,i+3)*x(j+2)+a(j+3,i+3)*x(j+3)
        enddo
        y(i+0)=t1
        y(i+1)=t2
        y(i+2)=t3
        y(i+3)=t4
      enddo
      return
      end

```

In the case of the unoptimized version, two array elements ( $a(j,i)$ ,  $x(j)$ ) have to be loaded into registers, resulting in two loads for one multiplication and one addition per loop iteration. In the unrolled case, 20 elements (16 elements of  $a$ , 4 of  $x$ ) have to be loaded into registers, resulting in 20 loads for 16 multiplications and 16 additions. This effect is particularly important on machines that can do more floating point operations than loads in one cycle. At the same time, we have eliminated dependencies and better exposed the instruction parallelism. Timing on an IBM 590 workstation shows that, for the unoptimized version, one cycle is needed per loop iteration, whereas in the optimized version only .27 cycles are needed, corresponding to a speed of 240 Mflops. This impressive performance number cannot be explained solely by the eliminated dependencies and improved parallelism. If we had two loads per loop iteration, the best we could expect would be .8 cycles. Our earlier observation (Table 1), that the higher level BLAS routines perform better than the lower level ones, is related to the fact that, by suitable loop unrolling one can obtain a better floating point to load/store ratio for the higher level routines than for the lower level ones.

In this simple case, loop unrolling was enough to obtain this speedup. Most compilers do a good job at unrolling when invoked at an appropriate optimization level.

## 5.4 Aliasing

Two arrays, labeled by different names, are aliased if they refer to identical memory locations. The rules for allowing certain kinds of aliasing are different in various programming languages. Fortran severely restricts aliasing, whereas C allows it. Hence, a Fortran compiler has more information than a C compiler and presumably can do a better optimizing job. In this section we will first explain the issues related to aliasing and then describe solutions for C and C++ programs.

Let us consider the following subroutine

```

      subroutine sub(n,a,b,c,sum)
      implicit real*8 (a-h,o-z)
      dimension a(n),b(n),c(n)

      sum=0.d0
      do 100,i=1,n

```



```

        a(i)=b(i) + 2.d0*c(i)
        sum=sum + b(i)
100    continue

        return
        end

```

According to the Fortran rules, two dummy arguments cannot be aliased if either one of them is modified by the subroutine. When calling a subroutine with identical arguments, the compiler will usually tolerate it, but the results are unpredictable. Such a subroutine call is syntactically correct, but semantically incorrect. We show below a semantically incorrect call to the subroutine and a semantically correct call. We can imagine other scenarios leading to the same aliasing. For example, if we equivalenced the arrays *a* and *b* in the main program, the semantically correct sequence would become incorrect.

```

        implicit real*8 (a-h,o-z)
        parameter (n=1000)
        dimension a(n),b(n),c(n)

        do 10,i=1,n
            a(i)=1.d0
            b(i)=1.d0
            c(i)=2.d0
10    continue

        c semantically correct call
        call sub(n,a,b,c,sum)
        c semantically incorrect call
        call sub(n,a,a,c,sum)

```

As we have learned already, most compilers do software pipelining. There are many possible ways to software pipeline the subroutine *sub*. One such possibility is shown below, in Fortran. For simplicity, let us assume that we are running on hardware capable of performing one floating point operation and one load or store per cycle. Let us also assume that the depth of the floating point unit is two stages for a fused floating point multiplication-addition, i.e. the result is only available two cycles after the operation started. The latency of load/store operations is one cycle. Under these assumptions, all the groups in the code below separated by comments, containing one or two instructions, can be performed in consecutive cycles. This is due to the fact that dependent groups are separated by at least one cycle. Therefore, three cycles only are needed to complete one loop iteration. As expected, the correct result is obtained using the semantically correct calling sequence. If the instructions are executed exactly in the order shown below, an incorrect result is obtained in the case of the semantically incorrect calling sequence. This is due to the fact that the old elements of *b* are used to form the sum, instead of the updated ones.

```

        tb=b(1)
        tc=c(1)
        do 100,i=1,n-1

c first cycle
            ta=tb+2.d0*tc
            tc=c(i+1)

```

```

c second cycle
    sum=sum+tb
    tb=b(i+1)

c third cycle
    a(i)=ta

100    continue
    i=n
    ta=tb+2.d0*tc
    sum=sum+tb
    a(i)=ta

```

Let us now consider a second possible way to schedule the instructions, as shown below. In this case the software pipelining is not optimal. In order to resolve the dependencies, we introduce an idle cycle, denoted by NOOP. Moreover, the floating point addition is not overlapped with a load/store. This loop iteration takes eight cycles, instead of three cycles in the software pipelined version. As the reader can verify, this non-optimal instruction schedule gives the correct result for both the semantically correct and the semantically incorrect Fortran calling sequences. Even though “NOOP”, is not a Fortran instruction we have mixed it with conventional Fortran syntax.

```

do 100,i=1,n
    tc=c(i)
    tb=b(i)
    ta=tb+2.d0*tc
    NOOP
    a(i)=ta
    tb=b(i)
    sum=sum+tb
100    continue

```

An important point is that a semantically incorrect structure in Fortran is perfectly correct in standard C or C++. According to the ANSI C standard, any two variables of the same type (such as integers or double precision numbers) can be aliased. This means that the compiler always has to assume the worst case scenario, the one in which severe aliasing is present. This is confirmed by looking at the assembler code generated by the IBM C compiler for the corresponding C version of the program. The instruction schedule generated is essentially identical to the one shown above. Obviously, the standard C conventions prevent the compiler from doing efficient, software pipelined, instruction scheduling. The execution time of the C program on an IBM 550 workstation is eight cycles per loop iteration, compared to three cycles per iteration for the Fortran code.

In the example above we concentrated only on the relatively simple case where  $a(i)$  was equivalenced to  $b(i)$ . More complicated aliasing schemes are possible in C, such as  $a(i)$  being equivalenced to  $b(i+2)$ . Scalar variables could also be aliased to certain array elements, such as  $a(9)$  being equivalenced to  $sum$ .

Most C compilers allow for compiler options or compiler directives that declare aliasing to be illegal. The details are vendor specific and can be found in the man pages. If these options are used, a C code should, in principle, be as fast as the corresponding Fortran code. The C compiler has then the same information and should be able to do essentially the same optimizations. In practice though, this is not always the case. Another possibility to alleviate the performance

penalty associated to aliasing is to assign by hand all array elements that are not subject to aliasing to different local scalar variables. In this way, the compiler can perform beneficial optimizations.

Even though aliasing problems exist mainly in C, they can also be found in Fortran, when indirectly indexed arrays or pointers are used. Let us look at an example involving indirectly indexed arrays.

```

subroutine sub(n,a,b,ind)
implicit real*8 (a-h,o-z)
dimension a(n),b(n),ind(n)

do 100,i=1,n
  b(ind(i))=1.d0+2.d0*b(ind(i))+3.d0*a(i)
100 continue

return
end

```

Evidently, consecutive loop iterations are not independent if  $ind(i) = ind(i + 1)$ . Fortran compilers have to take this possibility into account and therefore turn off software pipelining which involves working on several independent loop iterations simultaneously. Some compilers allow for the use of directives indicating that  $ind(i)$  is an invertible mapping and hence aliasing is not present.

Pointers are a standard in Fortran90. They are another feature which can lead to aliasing ambiguities. We will illustrate this effect for the sample Fortran90 program shown below that updates the positions of a set of  $n$  particles using their velocities. As discussed earlier, the ideal data structure would be one where the x, y and z components of all the arrays are adjacent in memory. In this example, we assume that the three components are in different arrays. Since the different elements of the position and the velocity are referenced by pointers only in the subroutine “move”, the compiler has to allow for the possibility that they are aliased. The generated code may not be efficient. Any arithmetic operation has to be preceded directly by the loads of all the operands involved and followed by an immediate store of the result.

```

MODULE declarations
  type position_type
    double precision, dimension(:), pointer :: rx, ry, rz
  end type position_type
  type velocity_type
    double precision, dimension(:), pointer :: vx, vy, vz
  end type velocity_type
END MODULE declarations

PROGRAM test
  USE declarations
  implicit none
  integer :: n
  type(position_type) :: position
  type(velocity_type) :: velocity

  n=1000
  ALLOCATE(position%rx(n),position%ry(n),position%rz(n))
  ALLOCATE(velocity%vx(n),velocity%vy(n),velocity%vz(n))
  position%rx=0.d0; position%ry=0.d0; position%rz=0.d0

```

```

velocity%vx=1.d0; velocity%vy=1.d0; velocity%vz=1.d0

call move(n,position,velocity)

END PROGRAM

SUBROUTINE move(n,position,velocity)
USE declarations
implicit none
integer :: i,n
type(position_type) position
type(velocity_type) velocity
double precision, parameter :: dt=1.d-1

do i=1,n-1,2
  position%rx(i)=position%rx(i)+dt*velocity%vx(i)
  position%ry(i)=position%ry(i)+dt*velocity%vy(i)
  position%rz(i)=position%rz(i)+dt*velocity%vz(i)
  position%rx(i+1)=position%rx(i+1)+dt*velocity%vx(i+1)
  position%ry(i+1)=position%ry(i+1)+dt*velocity%vy(i+1)
  position%rz(i+1)=position%rz(i+1)+dt*velocity%vz(i+1)
enddo

END SUBROUTINE move

```

## 5.5 Special functions

The calculation of special functions, such as divisions, square roots, exponentials and logarithms requires anywhere between a few dozen cycles up to hundreds of cycles. This is due to the fact that these calculations have to be decomposed into a sequence of elementary instructions such as multiplies and adds. The first advice is to keep the number of special function calls to a strict minimum. The number of special function calls can sometimes be reduced by storing the values of repeatedly used arguments in an array, instead of recalculating them every time. Frequently, usage of mathematical identities can reduce the number of special function evaluations. For example, calculating  $\log(x*y)$  is nearly two times faster than calculating  $\log(x)+\log(y)$ .

A large fraction of the CPU time in the evaluation of special function goes into the calculation of the last few bits. Relaxing accuracy demands has the potential of considerably speeding up these calculations. Most vendors have libraries that calculate special functions with slightly reduced accuracy, but significantly faster.

## 5.6 If statements

If statements slow down a program for several reasons. First, the compiler can do fewer optimizations in their presence, such as loop unrolling. Second, the evaluation of the conditional takes time by itself. Third, the continuous flow of data through the pipeline is interrupted when branching. The time for executing the branch condition itself is actually negligible, and/or overlapped by other instructions. In many cases, if statements can be significantly reduced or even eliminated completely by restructuring the program. This restructuring is very much context dependent, so that it is difficult to provide general guidelines on how to do it.

## 5.7 Loop overheads

Loops involve certain overheads. Registers needed within the loop need to be freed by storing their old values in memory. Other registers containing loop counters and addresses have to be set. The exit from the loops necessarily involves a mispredicted branch. The resulting overhead can vary from a few up to a few dozen cycles. Such overhead is obviously negligible if a lot of work is done within the loop, either because one iteration of the loop involves a lot of numerical operations or because many iterations through the loop are present.

## 5.8 Copy overheads in Fortran90

In Fortran90, it is possible to work on substructures of arrays. For example, a row of a matrix can be considered as a vector. In principle a good compiler can avoid, in most cases, copying the substructure into a work array. Unfortunately many compilers frequently do unnecessary copies, requiring superfluous operations as well as additional memory.

# 6 Optimization of memory access

As it has been stressed already several times in this article memory access is the major bottleneck on machines with a memory hierarchy. Therefore, optimizing the memory access has the largest potential for performance improvements. While floating point optimization can speed up a program by a factor two for in-cache data, memory access optimization can easily lead to performance improvements by a factor of ten or more for out of cache data.

In this section we discuss issues pertinent to memory access optimization. For readability, many examples will present loop structures that were not unrolled or otherwise optimized according to the principles put forward in the previous section. When timings are presented, the floating point optimizations are done by hand or by invoking the compiler with appropriate options.

## 6.1 Loop reordering for optimal data locality

The following code sequences differ in their loop ordering only:

	dimension a(n,n),b(n,n)		dimension a(n,n),b(n,n)
C	LOOP A	C	LOOP B
	do 10,i=1,n		do 20,j=1,n
	do 10,j=1,n		do 20,i=1,n
10	a(i,j)=b(i,j)	20	a(i,j)=b(i,j)

Let us discuss the memory access patterns of these two loops for the case of small and of big matrices. To simplify the analysis we assume that the size of the cache is very small, 128 double precision numbers only. In this case, a matrix is small when  $n$  is less than eight words, and therefore both  $a$  and  $b$  fit in cache. The length of a cache line is assumed to be eight doublewords. According to the Fortran convention, the physical ordering of matrix elements in memory is the following:

$a(1,1), a(2,1), a(3,1), \dots, a(n,1), a(1,2), a(2,2), a(3,2), \dots, a(n,2), \dots$

We refer to this access pattern as “column major order”. In C, the storage convention is just the opposite, “row major order”, meaning that the last index is “running fastest”. Therefore, the physical indexing of our small ( $n=8$ ) and big ( $n=16$ ) matrices is the following, in Fortran:

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255
16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256

In the following discussion we assume that the first matrix elements of both  $a$  and  $b$  are aligned on a cache line boundary, i.e. the first element of each matrix is also the first element of a cache-line. We also assume that all the data are out of cache at the beginning of the calculation. We distinguish the following cases:

- Loop A, small matrices

In this case the matrix elements will be accessed in the following order:

$$x(1,1), x(1,2), x(1,3), \dots, x(1,8), \quad x(2,1), x(2,2), x(2,3), \dots, x(2,8), \dots$$

where  $x$  denotes either  $a$  or  $b$ . However, this is not the physical order in memory. The elements  $x(1,i)$  and  $x(1,i+1)$  are actually eight doublewords apart and thus a cache miss will occur on the first eight loads of both  $a$  and  $b$ , as we access the physical memory locations 1, 9, 17, 25, 33, 41, 49 and 57. Since the cache can house all of the  $2 \times 8^2 = 128$  doublewords that were brought in during the first eight iterations of the double loop, all subsequent memory references will be cache hits. No other cache misses will occur in all of the remaining loop iterations. The load of all the  $2 \times 8^2$  elements involved  $2 \times 8$  cache misses, i.e. one cache miss for every 8 words.

- Loop A, big matrices

The memory access pattern will be the same as above, i.e. consecutive memory references are not adjacent. In this case they are 16 doublewords apart. After the first 16 iterations of the loop (each of which caused a cache miss) we will have loaded  $2 \times 16$  cache lines, i.e.  $2 \times 16 \times 8 = 128$  doublewords. The cache will be full. All of the subsequent cache lines loaded will need to overwrite the existing ones. The old cache lines will be flushed out of cache and stored back in the main memory. The cache line holding elements 129

to 136 will replace the cache line holding elements 1 to 8, the cache line holding elements 145 to 152 will replace the cache line holding elements 17 to 24, and so on. In the second iteration of the outer loop, the array element  $x(2, 1)$ , which was loaded in cache along with  $x(1, 1)$ , will no longer be available, as its cache line was replaced. This implies that the number of cache misses is now  $2 \times 16^2$ , i.e. one cache miss for every word accessed.

- Loop B

In this case we access the matrix elements exactly in the order in which they are stored in memory. We will then have one cache miss for every 8 doublewords for both small and large matrices. All the data brought in cache by the cache misses will be reused by the subsequent inner loop iterations.

We see that, for large matrices, the loop structure A results in a significant performance loss. For a realistically scaled up example, the performance degrades roughly by a factor of 30 on an IBM 590 when using loop A instead of loop B. Loop B gives the best data locality and is always to be preferred, even though in the case of small matrices the performance is the same for both orderings. The above conclusion remains valid in the more stringent, but realistic case, where the matrices are not aligned on a cache line boundary.

This example was chosen for didactic reasons only. In practice, for array copying, using the BLAS DCOPY routine is recommended. By using the calling sequence

```
call DCOPY(n*n,b,1,a,1)
```

the matrices a and b are considered as one-dimensional vectors of length  $n^2$  and the copying is done optimally from a data locality standpoint. At high optimization levels good performance can be obtained without BLAS, if the compiler collapses the two loops into a single one.

## 6.2 Cache thrashing

As explained in section 2.4, cache thrashing occurs if the effective size of the cache is much smaller than its physical size because of the constraints of the mapping rules. In this section we analyze cases where cache thrashing can occur in codes. Let us look at the following program.

```

program cache_thrash
  implicit real*8 (a-h,o-z)
c array x without buffer
  parameter(nx=2**13,nbuf=0)
c array x with buffer
C   parameter(nx=2**13,nbuf=81)
  dimension x(nx+nbuf,6)

  nl=2**10
  call sub(nl,x(1,1),x(1,2),x(1,3),x(1,4),x(1,5),x(1,6))
end

subroutine sub(n,x1,x2,x3,x4,y1,y2)
  implicit real*8 (a-h,o-z)
  dimension x1(n),x2(n),x3(n),x4(n),y1(n),y2(n)

  do 15,i=1,n-1,2
```

```

        y1(i+0)=x1(i+0)+x2(i+0)
        y1(i+1)=x1(i+1)+x2(i+1)
        y2(i+0)=x3(i+0)+x4(i+0)
        y2(i+1)=x3(i+1)+x4(i+1)
15      continue

      return
    end

```

On an IBM 590, with *nbuf*=0 we get a performance of just 3.5 Mflops, whereas 67 Mflops is achieved with *nbuf*=81 or any other reasonable nonzero value. Note that we did not modify at all the subroutine that is doing the numerical work. The reason for the extremely poor performance, in the first case, is that all six memory references are mapped to the same four slots in cache (Figure 3). So, even though all the data of size  $6 \times 2^{10} = 6144$  words that are accessed in the subroutine could easily fit in cache, the effective cache size in this case is only four cache lines ( $4 \times 32 = 128$  doublewords), which is not enough to hold all of the six cache lines. This example is of course contrived. In most applications the starting elements of these six arrays will not be separated by a high power of two. However, there are many algorithms, most notably Fast Fourier transforms, fast multipole methods, multigrid methods and wavelet transforms, where the leading dimensions are typically high powers of two. Padding the arrays will fix the performance problem in these cases. It is clear that the likelihood of running into cache thrashing is higher for directly mapped caches than for set associative ones.

On a computer with several cache levels, cache thrashing can occur for each level as well as for the TLB. In dealing with this, we recommend the conservative assumption that all memory levels behave like inclusive caches. This means that strides (usually leading dimensions) need to be adjusted in such a way that conflict cache misses are avoided for all cache levels.

Even in the absence of arrays with pathological cache behavior, there will seldom be perfect mapping. Good usage of the cache size is particularly difficult in the case of directly mapped caches. In this case, in order to increase the effective cache size, the arrays may have to be aligned by hand in memory. This can be done by copying them in a work array, and choosing the starting positions of the sub-arrays in the work array in an optimal way. An alternative is to align them in a common block. The first method has the advantage that the starting positions can be chosen dynamically. For the program *cache\_thrash* this first solution can be implemented as follows:

```

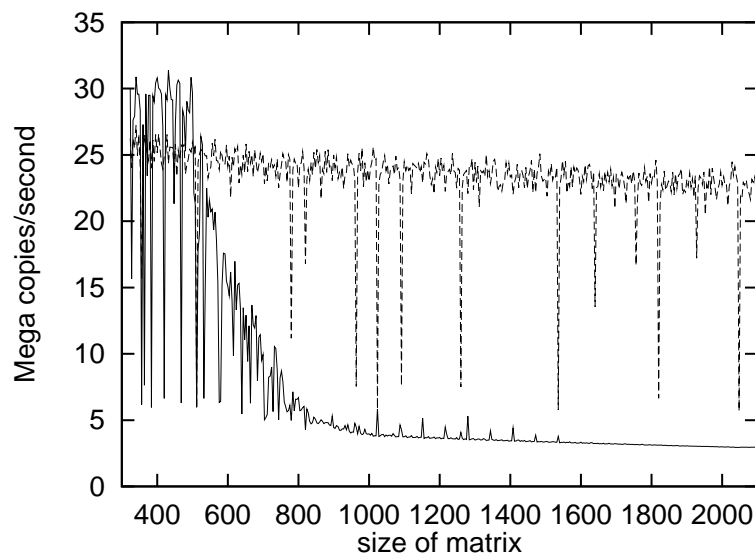
implicit real*8 (a-h,o-z)
parameter(nn=1024,nx=2**15)
dimension w(nn*6),x(nx,6),ist(6)

nl=....
x=.....

iist=1
do j=1,6
  ist(j)=iist
  do i=1,nl
    w(iist)=x(i,j)
    iist=iist+1
  enddo
enddo

```





**Fig. 4:** Performance of a matrix transposition with (dashed line) and without (solid line) blocking. In absence of blocking the performance decreases dramatically for large matrix sizes. The performance of the blocked version stays fairly constant with the exception of certain matrix sizes where cache thrashing occurs.

```

      call sub(nl,w(ist(1)),w(ist(2)),w(ist(3)),
&          w(ist(4)),w(ist(5)),w(ist(6)))
      end

```

### 6.3 Square blocking

Square blocking (or tiling) is a strategy for obtaining spatial data locality in loops where it is not possible to have small strides for all referenced arrays. One simple example is a matrix transposition, performed by the program below.

```

      subroutine rot(n,a,b)
      implicit real*8 (a-h,o-z)
      dimension a(n,n),b(n,n)

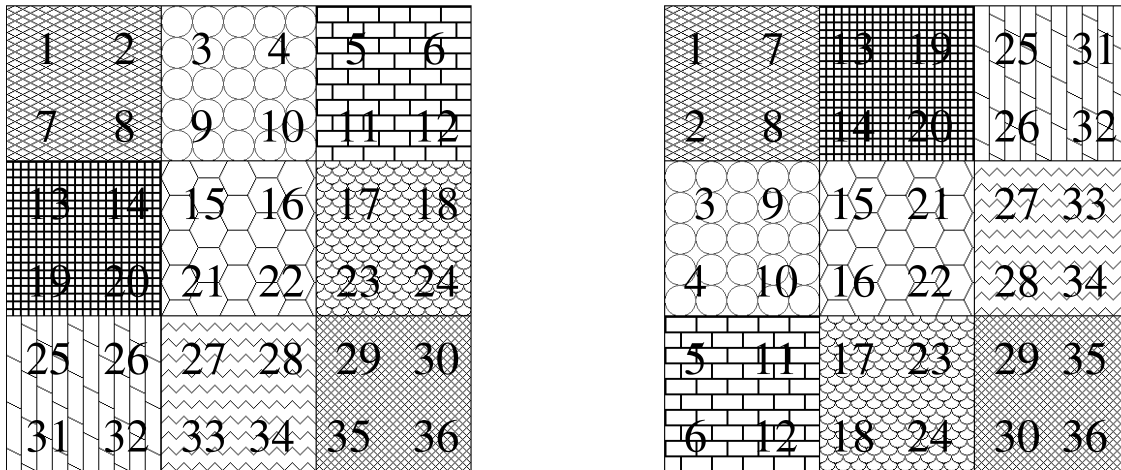
      do 100,i=1,n
      do 100,j=1,n
         b(j,i)=a(i,j)
100    continue

      return
      end

```

Obviously, it is not possible to have unit stride access for the elements of the arrays  $a$  and  $b$  at the same time. For large data sets, that do not fit in cache, the performance degradation is significant, as shown in Figure 4.

The fact that reasonable performance is achieved for small data sets suggests the solution for large data sets. Instead of transposing the matrix in one big chunk, we subdivide it into smaller sub-matrices and we transpose each of the smaller arrays. This divide-and-conquer strategy



**Fig. 5:** Schematic representation of a blocked matrix transposition. The matrices are first subdivided into sub-matrices (in this case of size 2 by 2) denoted by differently hashed backgrounds and then each pair of sub-matrices is transposed.

is called blocking and is shown schematically in Figure 5. The sub-matrices are indicated by different hashing patterns.

The Fortran implementation of the blocked version is the following:

```

subroutine rotb(n,a,b,lot)
implicit real*8 (a-h,o-z)
dimension a(n,n),b(n,n)

c loop over blocks
do 100,ii=1,n,lot
do 100,jj=1,n,lot
c loop over elements in each block
do 100,i=ii,min(n,ii+(lot-1))
do 100,j=jj,min(n,jj+(lot-1))
b(j,i)=a(i,j)
100 continue

return
end

```

The blocking parameter *lot* depends on the cache size. If the full physical cache were available, then *lot* would be chosen such that  $2 \times lot^2 = \text{the cachesize}$ . Because of the mapping rules previously discussed, the effective cache size is smaller than the physical cache size and we have to choose a smaller value of *lot*. The exact value could be determined by using subroutine “cache\_par”. Simply taking the effective cache size as equal to half the physical cache size works reasonably well for most matrix sizes, but cache thrashing occurs for some values, as shown in Figure 4.

An interesting question is whether it is necessary to square block with respect to several levels of the memory hierarchy, such as for the L1 cache, L2 cache and the TLB. According to our experience that is usually not necessary. L1 caches are usually small and blocking for them will lead to poorly performing short loops. Blocking for the TLB is necessary only for very

large data sets. In addition, the number of TLB misses is, in general, negligible compared to the number of cache misses. In the case of a three-level memory hierarchy, it is only necessary to block for the L2 cache. The new IBM Power3 architecture, with large L1 and L2 caches, might be an exception from this rule.

Blocking is error prone and the code is likely to become less legible. Most compilers do square blocking when invoked with certain options. We verified that the IBM and SGI compilers generate optimal blocking, if invoked appropriately, in this easy example. For more complicated loops, compilers are unlikely to do a satisfactory blocking job.

## 6.4 Line blocking

Square blocking is based on a static picture. A big matrix is subdivided into smaller rectangular matrices, on which we work sequentially. Square blocking causes a doubling of the number of loops. Line blocking, also called row- or column-oriented blocking is based on the understanding of the dynamics of the data flow through the cache. Whereas in simple blocking rectangular (or square) blocks are utilized, whose size is related to the size of the cache, irregular domains (Figure 8) can be used in line blocking, as will be explained in the following. The advantage is that fewer loops are needed and, in general, the innermost loops are longer.

To derive line blocking let us start with the blocked version of the matrix transposition subroutine *rotb*. Evidently, the algorithm is independent of the order in which we visit the different blocks. Hence, we can switch the order of the *ii* and *jj* loops. In addition, we can also merge the *ii* and *i* loops to obtain:

```

subroutine rots(n,a,b,lot)
implicit real*8 (a-h,o-z)
dimension a(n,n),b(n,n)

do 100,jj=1,n,lot
do 100,i=1,n
do 100,j=jj,min(n,jj+(lot-1))
    b(j,i)=a(i,j)
100 continue

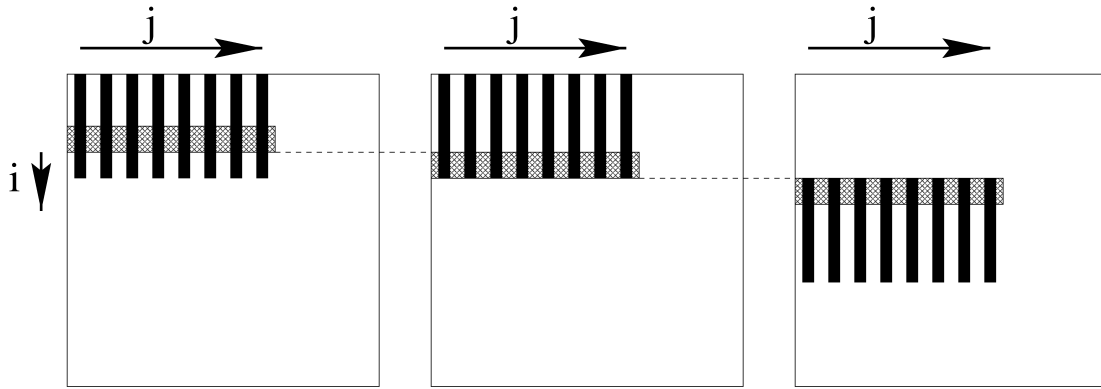
return
end
```

Let us first consider the simplest, but unlikely, case where the leading dimension *n* is a multiple of the cache line length. For the moment we will only concentrate on the data access pattern for the array *a*, since the array *b* has the best spatial data locality due to its stride one access. The array elements of *a* needed at different stages of transposition, as well as their location in cache, are shown in Figure 6.

The storage locations for the elements of *a* in a directly mapped cache are shown in Figure 7. Matrix elements that are brought in at a certain stage, without being used immediately, are used in subsequent iterations before the cache line holding these elements is overwritten by other cache lines.

In general, the dimension of the matrix is not a multiple of the cache line size. The cache lines that are needed in this case are shown in Figure 8.

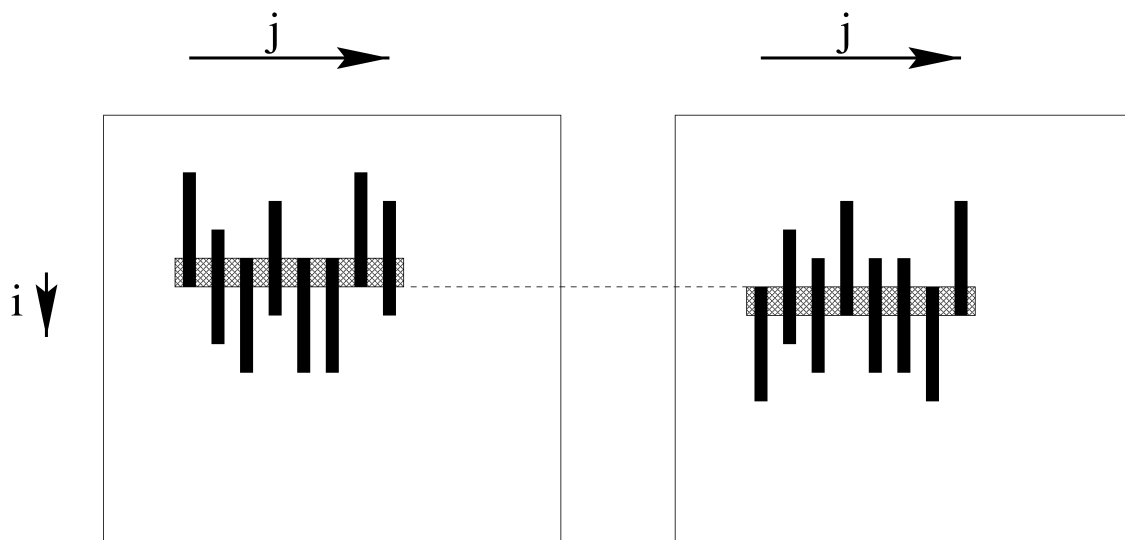
The choice of the blocking parameter *lot* was obvious for the data set defined in Figure 6 and a cache whose characteristic parameters were declared in Figure 7. In realistic cases, the determination of the largest possible value of *lot* is more difficult and is best done by experimentation.



**Fig. 6:** Areas of the matrix  $a$  that are used during three successive iterations of the  $i$ -loop in the subroutine “rots”. The leading dimension  $n$  of  $a$  is taken to be twelve, the blocking parameter  $lot$  is eight. A cache line holds four doublewords. The elements accessed for fixed values of  $i$  are denoted by the hashed horizontal bars. The cache lines to which these elements belong are indicated by black vertical bars. Since “rots” is written in Fortran, the matrix is stored in column major order.

$i=1$	1,1	1,4	1,7	1,2	1,5	1,8	1,3	1,6
$i=2$	2,1	2,4	2,7	2,2	2,5	2,8	2,3	2,6
$i=3$	3,1	3,4	3,7	3,2	3,5	3,8	3,3	3,6
$i=4$	4,1	4,4	4,7	4,2	4,5	4,8	4,3	4,6
$i=5$	5,1	5,4	5,7	5,2	5,5	5,8	5,3	5,6

**Fig. 7:** Cache locations of the elements of  $a$  in the subroutine “rots”. As in Figure 6, the leading dimension  $n$  of  $a$  is taken to be twelve, the blocking parameter  $lot$  is eight. The first five iterations of the  $i$  loop are depicted. The order in which the elements are accessed in the  $j$  loop is obtained by reading the elements in the usual order, i.e. from left to right and then downwards. The figure assumes that one cache line (depicted as a box) can hold four array elements and that the entire cache (directly mapped) can hold eight cache lines. We note that all cache lines are occupied after eight iterations of the  $j$  loop. Of the 32 words loaded during the first  $i$  iteration, only eight are used in the same iteration. However, all the other 24 array elements are used in the following three  $i$  iterations. No other cache lines have to be loaded during these three  $i$  iterations. The elements are overwritten after four iterations of the  $i$  loop, but then they are no longer needed. In absence of the  $j$  loop blocking, these 24 elements would be overwritten before used.



**Fig. 8:** Areas of the matrix  $a$  that are used during two successive iterations of the  $i$  loop in the subroutine “rots”. The leading dimension  $n$  of  $a$  is not a multiple of the cache line size. The symbols used for the cache lines and elements accessed are the same as in Figure 6.

The performance of a matrix transposition done in this way is shown in Figure 9. Even though cache thrashing is prevented in this case, we observe certain large matrix sizes where serious performance degradation occurs. This is due to TLB thrashing. Since the TLB behaves like a higher level cache, we can use the same simulation program “cache\_par” to determine leading dimensions  $nd$  and blocking parameters  $lot$  that avoid both cache and TLB thrashing, as shown below. In contrast to square blocking, line blocking with respect to several cache levels does not lead to a larger number of loops. The number of loops is the same as for blocking with respect to a single level, only the leading dimension has to satisfy more constraints.

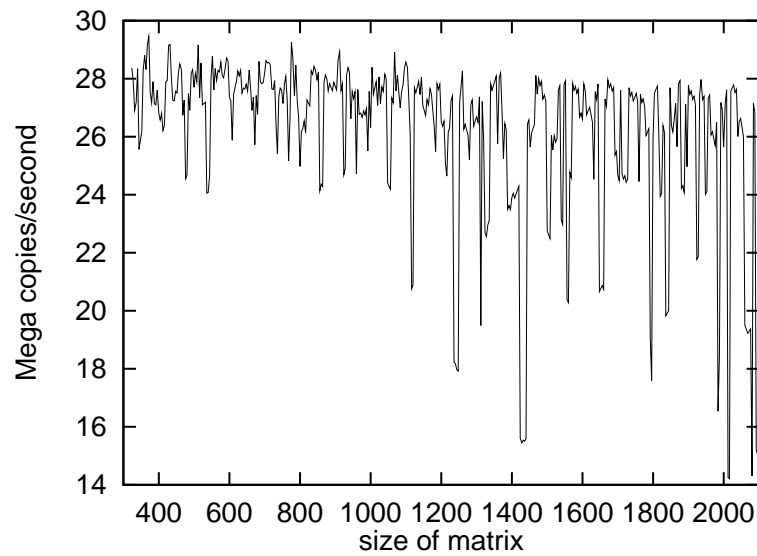
```

      nd=n
111  continue
      call cache_par(ncache_line,ncache_size,nd,lotc)
      call cache_par(ntlb_line,ntlb_size,nd,lott)
c if we have frac of the physical cache and tlb size, we are satisfied
      frac=0.75d0
      if (lotc.ge.frac*ncache_size/ncache_line .and.
        &    lott.ge.frac*ntlb_size/ntlb_line) then
          goto 222
      endif
      nd=nd+1
      goto 111
222  continue
      lot=min(lotc,lott)

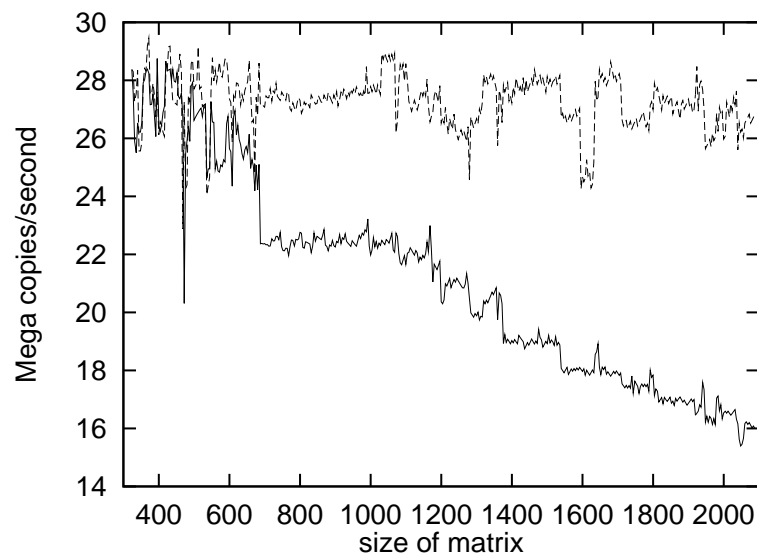
```

Once  $nd$  and  $lot$  are optimized for cache and TLB, we get good performance for all matrix sizes, as shown in Figure 10.

The performance data shown in Figures 4, 9, and 10 were actually obtained with unrolled versions of the subroutine “rots”. Neglecting the tail section, which would insure correctness for odd values of  $n$ , this code has the following form:



**Fig. 9:** Performance of a matrix transposition using one-dimensional blocking with an optimally adjusted leading dimension and a value of *lot* calculated by the cache simulation program “*cache\_par*”. For certain matrix sizes, serious performance degradation occurs due to TLB thrashing.



**Fig. 10:** The dashed lines show the performance of a matrix transposition using one-dimensional blocking where *nd* and *lot* were optimized with respect to both cache and TLB. The solid lines show the performance of the “*DGETMO*” matrix transposition subroutine from the ESSL library for the same leading dimensions *nd*.

```

do 100, jj=1, n, lot
do 100, i=1, n-1, 2
do 100, j=jj, min(n, jj+lot-1), 2
    b(j+0, i+0)=a(i+0, j+0)
    b(j+0, i+1)=a(i+1, j+0)
    b(j+1, i+0)=a(i+0, j+1)
    b(j+1, i+1)=a(i+1, j+1)
100 continue

```

The fact that in this unrolled version we are referencing the array elements  $a(i+0, j+0)$  and  $a(i+1, j+0)$  suggests that we might actually use data belonging to two adjacent cache lines. The referenced array elements could even belong to two adjacent pages, an unlikely possibility which we choose to neglect. In all the tests presented in this section, we have determined  $nd$  and  $lot$  such that one associativity class of the cache can be filled up to a certain fraction of its physical capacity. In the case that two adjacent cache lines are accessed in one loop iteration, the second cache line can always be stored in another associativity class. The remaining two associativity classes are, roughly speaking, reserved for the two data streams of the input array  $b$  ( $b(*,i)$  and  $b(*,i+1)$ ), each having a stride one access pattern. Using two associativity classes for the input array  $b$  seems to be a waste. Its cache lines could, in principle, be overwritten immediately after being used. However, since we have no influence over which elements are overwritten (the least recently used policy in general determines that), we have to let these array elements age sufficiently in cache before they can be overwritten by useful new data. It is obvious that for this kind of memory access optimization, it is very convenient to have a larger number of associativity classes, such as the four-way associative cache of the IBM Power2 series that was used for these tests.

The optimization techniques presented here are well beyond what a compiler can do. No compiler can determine optimal leading dimensions  $nd$ , nor change the leading dimensions in a program. Similarly, no compiler can determine the blocking parameter  $lot$ , since the leading dimension may not be known at compile time.

The performance of one-dimensional blocking is limited by two factors. Since we have transformed a data access pattern with no locality into one with spatial data locality, the performance cannot exceed the stride one performance. The best possible performance is reached depending upon whether the innermost loop is long enough, i.e. the blocking parameter  $lot$  is big enough. The maximum value of the blocking parameter with respect to one memory hierarchy level is given by the number of slots available on that level, by which we mean the number of basic units such as cache lines or pages that that level can accommodate. The overall blocking parameter  $lot$  equals the smallest blocking parameter of all the cache levels. On an IBM 590, each associativity class of the cache and the TLB has 256 slots. Loops of length 256 give reasonably good performance. For matrix transposition, the performance shown in Figure 10 is close to the one obtained for stride one data access. On the Digital AU433, the TLB has only 64 entries which leads to loops too short for good performance in matrix transposition.

## 6.5 Prefetching

As mentioned repeatedly in this article, frequently, the CPU is not fed fast enough with data from memory. There are two possible reasons that can give rise to this bottleneck. The first reason is that the memory bandwidth is not large enough for data to arrive at the required rate from main memory. If a program schedules one load per cycle but the bandwidth to main

memory only allows one transfer every two cycles, that program will be slowed down by a factor of two. Even if the bandwidth allows for the transfer of one item per cycle, there is a second reason why programs can be slowed down, the memory latency. If load instructions are scheduled briefly before the data item is needed, possible cache misses can cause the CPU to idle until the data arrives in registers. To avoid this effect, the data item has to be requested long before needed. If the time between the initial request and the use of the data can be bridged with other computations for which the operands are available, the CPU will not idle and good performance is obtained. To a certain extent the compiler tries to schedule loads way ahead. However, there are limits to what the compiler can do, mainly related to the number of available registers. If a data item is loaded long before it is needed, it will occupy a register during this entire period. The solution would be an instruction that transfers one or several cache lines from the main memory into cache. In this case, it is no longer necessary to schedule the load way ahead, but only slightly earlier (as explained in section 5.2) since the data will be in cache and no cache misses will occur. An instruction of this type is called a prefetch. Prefetch instructions are part of several instruction sets. Prefetch instructions are not provided for in the specifications of commonly used programming languages.

Alternatively, prefetching can be implemented in hardware, for example by using stream buffers. Based on runtime information from the first few iterations of a loop, a prediction is made as to what cache line will be needed in subsequent iterations. These cache lines are then preloaded into the stream buffer from which they can be transferred very rapidly in cache. Stream buffers work well for simple access patterns only, such as small and constant stride access.