INTRODUCTION

These notes, used in a two-part four-hour short course, introduce the reader (mostly the scientific programmer) to some of the main scalar optimization concepts and techniques associated with modern superscalar architectures. The hope is that an understanding of these techniques and basic hardware issues will enable the new comer to this area to intelligently use the compiler’s (f90, f77 or cc) optimization options, in combination, perhaps, with direct manual efforts. All of the illustrative examples contained here were tested on the Texas A&M University SGI Origin3000 System, which is based on the MIPS R14000 CPU.

Additional software and hardware material that is needed as a background is covered during the presentation in a classroom setting. Such basic hardware/architecture concepts are well within the reach of the average programmer. Their understanding is important in dealing with code optimization issues. In particular, the understanding of two concepts is vital: that of locality of reference and its relationship to the function and efficient use of the memory hierarchy, on the one hand, and, on the other, that of pipelining. Important as well is to remember the approximate performance values of some of the hardware. These notes supply this information only in regards to the MIPS R14000 CPU.

In reading these notes please bear in mind the following:

- All “sample” program performances reflect late 2001 f77 compiler technology on a SGI Origin3800. A keen user should recompile each sample program with Fortran 90 and run these code”lets” on the SGI Origin3800.

- I call the second level (L2) cache throughout the notes by three or four different names: Gcache (for global cache), secondary cache, unified secondary cache, and L2 cache.

- I use interchangeably the terms for CPU cycle, clock period (CP), clock cycle, or just cycle.

- All sample code showing was executed on the 500 MHz R14000 CPUs (not the 196 MHz ones) of a 48-CPU Origin3800 system. The TAMU Origin3800 has 48 500MHz CPUs.

- I use interchangeably the terms R14000 and R14K.

- A memory word here always refers to 64 bits, unless stated otherwise.

- K2, the 48-CPU Origin3800, has R14K’s of 500Mhz. with a 8MB L2 Caches. These R14K’s should yield a performance improvement of 10%-50% over that of the 195Mhz version.

Spiros Vellas, Sr. Analyst, CIS, TAMU
### BASIC FACTS, TERMS, and IDEAS

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Period/Cycle (CP)</td>
<td>Basic unit of time by which all operations in a CPU are timed and synchronized. The SGI R14K’s on the Origin3000 have a CP of 2.0 nanoseconds.</td>
</tr>
<tr>
<td>Functional Unit</td>
<td>A hardware device that performs an elementary computer operation, such as addition. Functional units receive and transfer data to registers only.</td>
</tr>
<tr>
<td>Register</td>
<td>Fastest but also smallest in capacity storage device, performs direct transfers to/from functional units from/to a memory. The <em>program counter</em> (PC) is a specially assigned register that holds the address of the currently executing instruction.</td>
</tr>
<tr>
<td>Cache</td>
<td>High-speed memory storage made of static RAM. Much smaller and faster than main memory. Holds copies of some of memory’s words. A cpu’s request for a main memory word results in a search for that word in the cache first. When not found there, the search is directed to lower level storage. Speeds (access time to registers) range from 1 to 15 cycles.</td>
</tr>
<tr>
<td>Basic (Code)Block</td>
<td>A section of code with only one entry (at the top) and only one exit (at the bottom). Used by compilers as basic organization and optimization unit.</td>
</tr>
<tr>
<td>Local Variable</td>
<td>Initially allocated on a stack (by default) in main memory, it is a variable whose scope is the subprogram it appears in. It cannot be a dummy or an actual argument, or a global variable. <strong>Compilers strive to keep scalar local variables in registers</strong>, and only when they run out of available registers do they <em>spill</em> such variables to lower level storage: cache or main memory.</td>
</tr>
<tr>
<td>Size Units</td>
<td>1KB=$2^{10}$B=$2^7$words=128words, 32KB=4096 words, 1MB=$2^{20}$B=$2^{17}$words=131,072words, 1GB=$2^{30}$B, 1TB=$2^{40}$B, 128B=16words. One word=64 bits (Double precision)</td>
</tr>
</tbody>
</table>
CODE OPTIMIZATION OBJECTIVES

• **Reduce CPU time for CPU-bound jobs by**
  • Exploiting space and time locality of reference by:
    • Improving cache and register reuse
    • Avoiding excessive branching (goto’s, if-then-else, sub calls)
    • Engaging concurrently the max. num. of pipes by:
      • Using the software pipelining (SWP) capability of the compiler, unrolling, and by appropriate grouping of terms
  • Other minor or very specific optimizations

• **Reduce I/O wait time for I/O-bound jobs by**
  • Making I/O requests transfer large data blocks
  • Making I/O requests asynchronous when substantial cpu-bound computation is possible between I/O requests
  • Merging long I/O Lists into one variable of contiguous allocation
  • Avoid/minimize use of formatted I/O

No further coverage I/O issues is pursued here.
LOCALITY OF REFERENCE PRINCIPLE

During execution, many programs exhibit a behaviour known as locality of reference. A program’s memory access pattern is said to have locality of reference when within a time window a certain address is accessed much more frequently than others and/or its access causes the accesses of nearby addresses as well.

Types of locality

- Temporal locality - Recently referenced items (data or instructions) are likely to be referenced again soon. This phenomenon is seen, for example, in programs with nested loops that execute the same instructions many times. Temporal locality, in this case, can be exploited by keeping recently accessed instructions in a convenient place (a cache) where they can be fetched quickly for decoding and execution.

- Spacial locality - tendency in a program to access items (data or instructions) whose addresses are near each other during a time window. For example, Fortran/C array columns/rows in matrix multiplication, stacks, tables, points in a mesh, etc. One can take advantage of spacial locality. For example, when the CPU issues a request to access a word from memory, one can in parallel with the requested word, retrieve into a cache, whereby data can be accessed swiftly, nearby words in anticipation that they too will be needed soon.

The 90/10 ratio

Many programs heavily use only a certain portion of their address space during any time interval. Empirically (according to Hennessy & Patterson [4]), it has been found that the typical program spends 90% of its execution time on only 10% of the code. For example, this occurs in nested DO/for-loop code segments. The validity of the locality of reference provided the basic rationale for introducing cache-based computer architectures with hierarchical memory structures.
R14000 MAJOR PERFORMANCE CHARACTERISTICS

- Superscalar (can issue up to 4 instructions per cycle)
- 500 MHz clock frequency (i.e., CP=2.0 nanoseconds) for the R14K’s on the Origin3000, and 196 MHz (i.e. 5.1 nanoseconds) for those R14K’s on the Origin3000.
- Prefetching
- Out-of-Order Execution
- $784 = 4/(5.1 \times 10^{-9})$ MIPS peak instruction issue rate
- 392 MFLOPS peak floating-point computation rate
- 1.2 Gbytes/sec peak data transfer rate from memory to cache.

Prefetching
Implemented with special hardware, this feature refers to the capability of the R14K to issue a *prefetch* instruction to fetch the next block of instructions in the execution sequence and place them in the secondary cache before they are actually needed. Therefore, when the R14K requires the next sequence, it can quickly fetch this sequence from the secondary cache and not from main memory. If this block, for whatever reason, is not needed, the area it occupies in the secondary cache is overwritten with other instructions. Prefetching enables the compiler to place early a given block of instructions, soon to be needed, as close to the CPU as possible.

Out-of-Order Execution
This feature refers to the capability of the R14K to execute instructions as soon as their operands (input values) become available regardless of the original instruction sequence. The R14K, in effect, rearranges (in a way that does not change the program’s semantics) the original instruction sequence so as to keep as many execution units concurrently busy as possible.
• Distributed Shared Memory (DSM). Memory is physically partitioned among the nodes but accessible by any CPU.

• The hardware enforces cache coherence across all nodes through a directory based scheme. The HUB device (crossbar switch) in each node is used for all inter- and intra-node communications

• Non-Uniform memory access. The remote-to-local ratio for memory references is at least 2:1. The remote-to-local memory delay for frequently-accessed data by a CPU is reduced by a mechanism that moves the data from remote memory to memory close to that CPU

• Virtual shared memory system
THE R14000 ARCHITECTURE

The R14K is a 4-way superscalar processor. It can fetch and decode, that is, up to four instructions per cycle. Each decoded instruction resident now in the instruction cache is appended to one of the three instructions queues: the integer queue, the floating-point queue, and the address (or load-store) queue. Each of these 16-entry queues can dynamically schedule and issue instructions to appropriate execution units. The queues determine the execution order based on the availability of the needed execution units. Instructions are initially fetched and decoded in order, but they can be executed and completed out-of-order, allowing the R14K to have up to 32 instructions in various stages of execution. The R14K can graduate (complete) up to 4 instructions per cycle.
**PIPELINED FUNCTIONAL UNITS**

Some functional units on the R14000 are pipelined and some are not. A pipelined unit is partitioned into independent hardware subunits, each specializing in completing within one CP (clock period or cycle) a specific phase of an operation. This operation can be an arithmetic algorithm or some other type of operation, such as getting a count of the number of bits that are set to 1 in a register. The algorithm that implements the operation breaks it down into a fixed number of sequential steps (or phases) that correspond one-to-one with the hardware subunits in the functional unit. In this manner, a pair of operands advancing into, for instance, the add floating point unit, “visits” successively each of its 2 segments, occupying each segment for one CP, to complete the operation in 2 CPs. The number of CPs needed to travel through a functional unit is called the **functional unit time**, or **pipeline latency**.

More importantly, because each segment in a pipelined unit is independent, the unit can process a new instruction each CP, before a previous instruction has completed. This feature of concurrently processing several instructions, each at a different stage of completion in a functional unit, is called **pipelining**. Each functional unit is referred to as a **pipe** because its operation resembles that of a mechanical pipe. In a pipelined operation, once the pipe is full, the output rate is one result per CP. The figure below shows a four-segment pipe which can work on four separate operands, once full.

Thus, assuming a steady supply of operands through the 32 floating-point registers, the total number of cycles required to complete n add instructions is $2 + (n-1)$. A non-pipelined add unit would have taken $2n$ cycles. The resulting speedup due to pipelining in this case is $2n/[2 + (n-1)]$. Hence, for a large n, the speedup is ~2.

The floating-point division, the floating-point square root, the integer multiply, and the integer divide functional units are NOT pipelined on the R14K.

![A four-segment pipe (e.g., Integer to floating point conversion on MIPS R14000). It takes 4 CPs to complete its operation on an pair of operands. But once full, the completion rate is one per CP.](image-url)
SOME R14000 SPECIFICATIONS

- Clock Frequency: 196 MHz (1 cycle = 5.10 nanoseconds)
- 32 64-bit floating-point and 32 64-bit integer registers
- Can issue up to 4 instructions per cycle from 5 independent execution units:
  1 Load/Store - FPload/FPstore/load, store/...; and/or
  1 ALU1 - add/sub/logical/shift/...; and/or
  1 ALU2 - add/sub/logical/set mul div/...; and/or
  1 FPADD - add/madd/sub/logical/neg/...; and/or
  1 FPMUL - mul/div/recip/sqrt rsqrt/madd/cond_move

Functional/Execution Units

Table 1: Instruction Latencies and Repeat Rates

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency (CPs)</th>
<th>Repeat Rate (CPs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Add, Subtract, Logical Ops, branches</td>
<td>1</td>
<td>1 (fully pipel’d)</td>
</tr>
<tr>
<td>Integer Load/Store</td>
<td>2 (L1 cache hit)</td>
<td>1 (fully pipel’d)</td>
</tr>
<tr>
<td>Integer Multiply (single/double&lt;sup&gt;a&lt;/sup&gt;)</td>
<td>5-6/9-10</td>
<td>6/10</td>
</tr>
<tr>
<td>Integer Divide (single/double)</td>
<td>34-35/66-67</td>
<td>35/67</td>
</tr>
<tr>
<td>Integer to Floating-point conversion (single)</td>
<td>4</td>
<td>1 (fully pipel’d)</td>
</tr>
<tr>
<td>Floating-point Load/store</td>
<td>3 (L1 cache hit)</td>
<td>1 (fully pipel’d)</td>
</tr>
<tr>
<td>Floating-point Multiply (single/double)</td>
<td>2</td>
<td>1 (fully pipel’d)</td>
</tr>
<tr>
<td>Floating-point Multiply-Add (single/double)</td>
<td>2/4</td>
<td>1 (fully pipel’d)</td>
</tr>
<tr>
<td>Floating-point Divide (single/double)</td>
<td>12/19</td>
<td>14/21</td>
</tr>
<tr>
<td>Floating-point Square Root (single/double)</td>
<td>18/33</td>
<td>20/35</td>
</tr>
<tr>
<td>Floating-point Recip. Square Root (single/double)</td>
<td>30/52</td>
<td>30/35</td>
</tr>
</tbody>
</table>

<sup>a</sup> Single/Double refer to 32-bit and 64-bit data respectively

Note that the **repeat rate** of an instruction is the number of clock periods that must transexpire before the same instruction can be reissued. Hence, the integer multiply and divide functional units are not pipelined, and neither are the floating-point divide, square root, or reciprocal square root units.
Note that by far the worst possible delays are those incurred when fetching data from main memory. This statement assumes that the occurrence of main memory misses (page faults) settled at the disk level are of negligible frequency.
SPECIFICATIONS OF THE CACHE HIERARCHY

Primary ICache (Instruction Cache, L1, on-chip, 32 KB)
- 4 instructions/cycle peak rate
- 2-way set associative
- 64-byte cache line/block
- 6-cycle minimum (miss) delay to transfer a cache line from the Gcache
- Subset of the Unified Secondary cache

Primary Data Cache (Dcache, L1, on-chip, 32 KB=4096 words)
- Non-blocking (Does not stall on a miss)
- 32-byte (4 64-bit words) cache line (or block)
- 2-way set associative (set_size = 2 * 4 = 8 64-bit words; Num of sets =512)
- 3-cycle minimum (miss) delay to transfer a cache line from the L2 cache
- Contains subset of data in L2 cache
- writeback (w.r.t L2 cache), LRU replacement policy

(Unified) Secondary Cache (L2, off-chip, 8MB=1,048,576 words)
- Non-blocking (Does not stall on a miss; permits upto 2 outstanding misses to main memory)
- 128-byte (16 64-bit words) cache line (or block)
- 2-way set associative (set_size = 2*16=32 64-bit words; Num of sets = 4096)
- 200-cycle minimum miss delay to transfer a line from main memory
- writeback type cache, LRU replacement policy
- Second level (L2) storage for instructions and data

General
- From the perspective of the L1 & L2 caches, main memory is subdivided in consecutive blocks (or cachelines). Each block itself comprises a series of consecutive addresses. The size of a block is typically 128 bytes (16 Dwords)
DATA ACCESS IN THE MEMORY HIERARCHY

An Example: A 2-way Set Associative Cache with k Sets

All transfers are in blocks (also called cache lines). The L2 cache of R14K is 2-way set associative. It is organized and is addressable in units called sets that are two blocks each (128B), and where the correspondence between memory blocks and cache sets obeys:

\[ \text{mem_blk_num \ Modulo (Num_Sets)} = \text{Cache_Set_Num} \]

A memory block that maps into a cache set can fit in either one of two slots that happen to be empty. When, during execution a memory block is needed and the set it maps to is fully occupied, then a conflict miss occurs, causing thereby the Least Recently Used (LRU) block in the set to be replaced by the new block. A cache miss occurs as well at the start of executing a sequence of code (cold-start-miss or compulsory miss) or when the currently needed data by an executing program do not all fit in the cache (capacity miss) due to having blocks discarded and later fetched. A 1MB cache has \(2^{20-8} = 2^{12} = 4096\) sets.

Writeback

If a block in a cache has been updated (i.e., it differs from its memory counter-part), it is not copied back to memory immediately unless there is a miss caused by another memory reference that maps to that same set, or until termination.

Non-Blocking

On a miss a non-blocking cache allows subsequent cache accesses to proceed. On the R14K up to two misses can be outstanding before the cache blocks other access.
DATA ACCESS IN THE MEMORY HIERARCHY

Data access patterns

Memory access patterns affect program performance directly. This is especially the case in cache-based systems. As far as performance is concerned the best access pattern is one where (sequentially) consecutive addresses are involved. When the access stride is 2, 3, or more the efficiency of the memory hierarchy is progressively reduced. Hence, data memory layout is very important. In Fortran, arrays, multidimensional arrays included, are laid out in column major fashion. That is, consecutive addresses correspond to traversing, say, a 2-dimensional array, by going down (or up) its columns. Hence, the stride involved in accessing array elements in the same row but from consecutive columns is equal to the size of a column. In C the storage format of arrays is by row major fashion, exactly the opposite of Fortran.

```fortran
REAL A(100,100)
DO I=1, 78            DO J=1, 78
   DO J=1, 11            DO I=1, 11
      A(I,J) = ...            A(I,J) = ...
   END DO                END DO
ENDO                  ENDO
```

The loop on the left accesses array A with a stride of 100 (baaad news...). On the right, the stride is 1. On the other hand, in C the loop on the left presents no problem, that on the right has un unfavorable access pattern. The 3-level loop below exhibits a more involved access pattern. The C(*) array during reads is accessed with a stride of 100, while the writes on the same array are done with a stride of 1. The B(*) array, on the other hand, is being read with a stride of 5000 (=100*50).

```fortran
REAL B(100,50,200), C(1000)
DO I=1, 20
   DO J=1, 50
      DO K=4, 100
         C(I) = C(1+(I-1)*100) + B(I,J,K)
      END DO
   END DO
END DO
```

In code development, one should strive to minimize the instances where the memory access pattern has stride other than 1.

In Fortran
In C

Large-memory programs
When programming on the Origin 2000, it important to keep in mind that its main memory is physically distributed: made up of many local memories. The latter are physically built-in in every constituent cpu node, which currently (Sept 1999) has 2 cpus. This aspect of the Origin 2000 system can have a negative impact on performance for scalar programs with memory needs exceeding the size of a node’s local memory (currently, 0.5Gb). The system in that case allocates the additional memory needed from the local memories of other nodes, thereby incurring additional overhead for data movement. Hence for large-memory programs, one might very well benefit from parallelizing such codes, other things being equal.
SPEEDUP & TIME MEASUREMENT

Speedup \[ S = \frac{T}{T_{\text{opt}}} \]

\( T_{\text{opt}} \) and \( T \) refer to measurements of elapsed CPU time for the execution of a program respectively with and without some optimization feature. One can also define scalar speedup in terms of flop (floating-point operation) rates: \( S = \frac{\text{flop rate}_{\text{opt}}}{\text{flop rate}} \). The latter definition, however, is appropriate only for codes where floating_point operations dominate the execution profile. Strictly speaking, the above definition of speedup in terms of cpu time is only appropriate for programs that are cpu-bound, that is, with a minimal I/O component. Or, equivalently, this speedup definition accounts for only the cpu-bound part of a program. When I/O is a non-negligible part of the execution profile, it is best to use the elapsed wall-clock time as the quantity to measure.

CPU Timing Utilities

Table 2:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>times a command. A UNIX utility</td>
</tr>
<tr>
<td>dtime</td>
<td>returns user and/or system cpu execution time. An IRIX Fortran function</td>
</tr>
<tr>
<td>cpu_time</td>
<td>returns user execution time. A Fortran 90/95 intrinsic.</td>
</tr>
<tr>
<td>clock()</td>
<td>Reports CPU time used. A C routine</td>
</tr>
</tbody>
</table>

All of the above routines are easy to use. For the details see their respective man pages. Most of the examples listed in these notes use the dtime routine. \textit{cpu_time} was not available at the time the examples were constructed.
PERFORMANCE PROFILING

- By PC-sampling (Program Counter)

\[
\text{ssrun -fpcsampx prog.exe} \quad \text{PID} \\
\text{prof -lines prog.exe.fpcsampx.m####}
\]

The monitored program, prog.exe here, is interrupted at a constant rate (once every 10 milliseconds) during execution and the value of the PC (program counter) is sampled/read. This sampling over time yields a histogrammic record of the frequencies by which various routines are accessed. It also enables the reliable estimation of the total amount of time a program spends in a routine. When ssrun executes prog.exe with PC-sampling enabled, the record of the sampling is stored in the binary file, prog.exe.fpcsampx####. The #### notation signifies the PID (process id) of the ssrun execution that the system attaches as a suffix to the generated prog.exe.fpcsampx file. The prof utility carries out the listing of the stored information.

Example

titan% ssrun -fpcsampx tim_opt0.exe < small.in

titan% prof -lines tim_opt0.exe.fpcsampx.m5236931

![SpeedShop profile listing generated Tue Jun 29 10:19:12 1999](image)

Function list, in descending order by time

<table>
<thead>
<tr>
<th>index</th>
<th>secs</th>
<th>%</th>
<th>cum.%</th>
<th>samples</th>
<th>function (dso: file, line)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>13.410</td>
<td>71.8%</td>
<td>71.8%</td>
<td>13410</td>
<td>DDPOWJHE (tim_opt0.exe: DDPOWJHE.f, 1)</td>
</tr>
<tr>
<td>[2]</td>
<td>0.772</td>
<td>4.1%</td>
<td>76.0%</td>
<td>772</td>
<td>LSAME (libblas.so: lsame.f, 1)</td>
</tr>
<tr>
<td>[3]</td>
<td>0.719</td>
<td>3.9%</td>
<td>79.9%</td>
<td>719</td>
<td>s_cmp (libfun.so: s_cmp.c, 63)</td>
</tr>
<tr>
<td>[4]</td>
<td>0.562</td>
<td>3.0%</td>
<td>86.5%</td>
<td>562</td>
<td>DLARTG (liblapack.so: dlarfg.f, 1)</td>
</tr>
<tr>
<td>[5]</td>
<td>0.527</td>
<td>2.8%</td>
<td>89.3%</td>
<td>527</td>
<td>DLAMCH (liblapack.so: dlamch.f, 1)</td>
</tr>
<tr>
<td>[6]</td>
<td>0.512</td>
<td>2.8%</td>
<td>92.1%</td>
<td>512</td>
<td>ZLASSR (liblapack.so: zlassr.f, 1)</td>
</tr>
<tr>
<td>[7]</td>
<td>0.186</td>
<td>1.0%</td>
<td>93.1%</td>
<td>186</td>
<td>ZSTEQR (liblapack.so: zsteqr.f, 1)</td>
</tr>
<tr>
<td>[8]</td>
<td>0.171</td>
<td>0.9%</td>
<td>93.9%</td>
<td>171</td>
<td>powdi (libfun.so: pow_di.c, 49)</td>
</tr>
<tr>
<td>[9]</td>
<td>0.147</td>
<td>0.8%</td>
<td>94.7%</td>
<td>147</td>
<td>COMPLIB_ZHEMV_U (libblas.so: zhemv_axpy.f, 259)</td>
</tr>
</tbody>
</table>
PERFORMANCE PROFILING

- By using the hardware event counters of R14K

```
perfex [-a | [-e event0] [[-e event1]]]
[-mp] [-x] [-y] command
```

```
f90/f77 [options] prog.f -lperfex
```

When `command` completes execution, perfex prints the values of various hardware performance counters. The R14K has two such counters available to monitor a number of events. The first counter handles any single event from the items numbered 0-15 below, while the second can handle any single event, 16-31. An average profile of all the events is provided using the -a, -x, and -y options together. One can also sample the event counters for specific segments of code. For that case, you link to your code the perfex library (-lperfex). See the BTEST1 program in these notes for an illustration. For more details see the perfex and R14K_counters man pages respectively. Notice that perfex as used in the first case yields data that represent a program as a whole during execution.

The integers event0 and event1 index this table:

<table>
<thead>
<tr>
<th>Integer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Cycles</td>
</tr>
<tr>
<td>1</td>
<td>Issued instructions</td>
</tr>
<tr>
<td>2</td>
<td>Issued loads</td>
</tr>
<tr>
<td>3</td>
<td>Issued stores</td>
</tr>
<tr>
<td>9</td>
<td>Primary instruction cache misses</td>
</tr>
<tr>
<td>10</td>
<td>Secondary instruction cache misses</td>
</tr>
<tr>
<td>15</td>
<td>Graduated instructions</td>
</tr>
<tr>
<td>21</td>
<td>Graduated floating point instructions</td>
</tr>
<tr>
<td>22</td>
<td>Quadwords written back from primary data cache</td>
</tr>
<tr>
<td>23</td>
<td>TLB misses</td>
</tr>
<tr>
<td>24</td>
<td>Mispredicted branches</td>
</tr>
<tr>
<td>25</td>
<td>Primary data cache misses</td>
</tr>
</tbody>
</table>
Scalar Code Optimization I

26 = Secondary data cache misses
:
31 = Store/prefetch exclusive to shared block in scache

Example

k2% f77 -R14000 -col120 -o test1.exe test1.f

k2% perfex -e 10 -e 26 test1.exe
-- TEST1: CPU SECONDS =  2.899008  FLOP_RATE =  11.7106E+06 --
   24223.83363400009   24223.83363400009
 Secondary instruction cache misses.................................... 1456
 Secondary data cache misses........................................... 65098

k2% perfex -e 25 test1.exe
-- TEST1: CPU SECONDS =  2.874945  FLOP_RATE =  11.8086E+06 --
   24223.83363400009   24223.83363400009
 Primary data cache misses............................................. 4407530
 Cycles........................................................................ 555081313

k2%

k2% f77 -R14000 -col120 -O3 -o test1.exe test1.f

k2% perfex -e 10 -e 26 test1.exe
-- TEST1: CPU SECONDS =  0.172688  FLOP_RATE =  19.6593E+07 --
   24223.83363400009   24223.83363400009
 Secondary instruction cache misses.................................... 1263
 Secondary data cache misses........................................... 20766

k2%

k2% perfex -e 25 test1.exe
-- TEST1: CPU SECONDS =  0.169805  FLOP_RATE =  19.9930E+07 --
   24223.83363400009   24223.83363400009
 Primary data cache misses............................................. 1036381
 Cycles........................................................................ 29288239

Note that in the -O0 case there are nearly 68 primary data cache misses for every 1 in the secondary cache, while the corresponding numbers for the -O3 case are 50 to 1. This makes for a 26.4% (18/68) relative improvement in the miss behaviour. And because of the high memory latency versus the L1_to_L2 latency, this improvement is very significant.

An example on the in-code use of the R14K_counters through the -lprefex option is provided by the BTEST1 sample program.

TLB & L2 Cache Misses

A high frequency of these events degrades performance dramatically. In particular, the TLB misses are in comparison extremely costly. For possible remedy see the -LNO compiler option (also see man lno).
PERFORMANCE PROFILING

- using hardware event counters by ssrun and prof

Commonly Used Events/Experiments:

- fgi_hwc: graduated instructions
- fcy_hwc: cycle counter
- fic_hwc: primary icache misses
- fisc_hwc: secondary icache misses
- fdc_hwc: primary dcache misses
- fdsc_hwc: secondary dcache misses
- ftlb_hwc: TLB misses
- fgfp_hwc: graduated floating-pt instrs

ssrun -exp event_name prog.exe
prof -lines prog.exe.event_name.m####

Example

titan% ssrun -ftlb_hwc tim_opt0.exe < small.in
titan% prof -lines tim_opt0.exe.ftlb_hwc.m4844033

SpeedShop profile listing generated Tue Jun 29 10:58:35 1999
prof -lines tim_opt0.exe.ftlb_hwc.m4844033
  tim_opt0.exe (n64): Target program
  ftlb_hwc: Experiment name
  hwc,23,23:cu: Marching orders
  R14000 / R19010: CPU / FPU
  48: Number of CPUs
  500: Clock frequency (MHz.)

  Experiment notes--
  From file tim_opt0.exe.ftlb_hwc.m4844033:
  Caliper point 0 at target begin, PID 4844033
  /home/s0v3474/howard/NEW/tim_opt0.exe
  Caliper point 1 at exit(0)

Summary of R14K perf. counter overflow PC sampling data (ftlb_hwc)--
6982: Total samples
  TLB misses (23): Counter name (number)
  53: Counter overflow value
370046: Total counts

Function list, in descending order by counts

<table>
<thead>
<tr>
<th>index</th>
<th>counts</th>
<th>%</th>
<th>cum.%</th>
<th>samples</th>
<th>function (dso: file, line)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>134885</td>
<td>36.5%</td>
<td>36.5%</td>
<td>2545</td>
<td>DDPOWJHE (tim_opt0.exe: DDPJHE.f, 1)</td>
</tr>
<tr>
<td>[2]</td>
<td>19186</td>
<td>5.2%</td>
<td>41.6%</td>
<td>362</td>
<td>ZLASR (libcomplib.sgimath.so: zlasr.f, 1)</td>
</tr>
<tr>
<td>[3]</td>
<td>15847</td>
<td>4.3%</td>
<td>45.9%</td>
<td>299</td>
<td>s_cmp (libftn.so: s_cmp.c, 63)</td>
</tr>
<tr>
<td>[4]</td>
<td>12084</td>
<td>3.3%</td>
<td>49.2%</td>
<td>228</td>
<td>COMPLIB_ZHEMV_U (libblas.so: zhemv.f, 259)</td>
</tr>
<tr>
<td>[5]</td>
<td>11395</td>
<td>3.1%</td>
<td>52.3%</td>
<td>215</td>
<td>ZLANHE (libcomplib.sgimath.so: zlanhe.f, 1)</td>
</tr>
<tr>
<td>[6]</td>
<td>10653</td>
<td>2.9%</td>
<td>55.7%</td>
<td>201</td>
<td>LSAME (libblas.so: lsame.f, 1)</td>
</tr>
<tr>
<td>[7]</td>
<td>9646</td>
<td>2.6%</td>
<td>57.7%</td>
<td>182</td>
<td>ZUNGZL (libcomplib.sgimath.so: zungz1.f, 1)</td>
</tr>
</tbody>
</table>
The advantage of using this form of profiling is that the information listed for the sampled event is per subroutine. One can, in combination with the results of perfex, estimate the time cost per subroutine for an instrumented event.
COMMON HIGH PERFORMANCE OPTIONS & CODE

**Fortran/C: -R14000 and -O3 options**

\[ f90/f77/cc \ldots -R14000 -O3 \text{ files} \]

The specification of the -r1000 and -O3 options should be the absolute least-effort version of your optimization actions.

**libfastm**

Contains faster, slightly lower-precision, versions of various routines: \texttt{sin}, \texttt{cos}, \texttt{tan}, \texttt{exp}, \texttt{log}, \texttt{pow}.

\[ f90/f77/cc \ldots -R14000 -O3 -lfastm \]

**Vector Intrinsics: acos, atan, asin, sin, cos, tan, exp, log, sqrt**

\[ f90/f77/cc -o \text{program files} -lm \]

**libcomplib.sgimath & libcomplib.sgimath_mp**

Contains: BLAS Levels 1,2, and 3, EISPACK, LAPACK, FFTs

In the compilations below, the \texttt{-mp} option causes the linking of the multiprocessing library, which, in turn, activates the parallel processing directives that may be present in the code:

\[
\begin{align*}
\text{f90/f77} & \ldots -l\text{complib.sgimath} -l\text{blas} -lfastm \\
\text{f90/f77} & \quad -mp \ldots -l\text{complib.sgimath_mp} -l\text{blas_mp} -lfastm
\end{align*}
\]

**libslatec & NAG**

General purpose mathematical libraries. For details type man slatec and man nag.

\[
\begin{align*}
\text{f90/f77} & \ldots -l\text{slatec} -l\text{blas} -lfastm \\
\text{f90/f77} & \quad -mp \ldots -l\text{slatec_mp} -l\text{blas_mp} -lfastm \\
\text{f90/f77} & \quad \ldots -l\text{nag} -lfastm
\end{align*}
\]

**Relevant man pages**

See man \texttt{complib}, man \texttt{3 math}, man \texttt{fft}, man \texttt{nag}, man \texttt{blas}, man \texttt{slatec}
A FIRST LOOK AT THE PERFORMANCE OF R14000

We are going to use matrix multiplication frequently to demonstrate incremental performance improvements. The -O3 optimization option encompasses a variety of subsidiary options that involve both hardware and software aspects, and it is the standard optimization option on the f90, f77 and cc compilers. To obtain a listing of the transformed code using the -R14000 & -O3 options, use the f77 -flist ... option.

```
PROGRAM TEST1                  ! HIGHSCHOOL MATRIX MULTIPLY ....
PARAMETER (N=257, NSQ=N*N)     ! WHEN N < 257 ALL OPS IN GCACHE
REAL*8 AA(N,N), BB(N,N), CC(N,N)
REAL*4 TM(2), TIME, FLOP_RATE, DTIME
DATA AA/NSQ*8.0099/, BB/NSQ*11.688/, CC/NSQ*0.000/
C       TIME=DTIME(TM(2))
C       DO I=1,N                        ! HIGH SCHOOL MATRIX MULTIPLY
            DO J=1, N
                    DO K=1, N
                            CC(I,J) = CC(I,J) + AA(I,K)*BB(K,J)
                    END DO
            END DO
        END DO
C       TIME=DTIME(TM(2))
        FLOP_RATE = (2*N*N*N)/TIME
        WRITE(6,FMT= '("-- TEST1: CPU SECONDS = ",F5.3," FLOP_RATE = ", + 2PE12.5," --")') TIME, FLOP_RATE
C       PRINT *, CC(20,12), CC(19,11)
END
```

```
k2% f77 -col120 -o test1.exe test1.f
k2% test1.exe
-- TEST1: CPU SECONDS = 2.022 FLOP_RATE = 16.7938E+06 --
   24060.26577840015 24060.26577840015

k2% f77 -col120 -R14000 -O3 -o test1.exe test1.f
k2% test1.exe
-- TEST1: CPU SECONDS = 0.045 FLOP_RATE = 76.0715E+07 --
   24060.26577840015 24060.26577840015
```

Results for N=557 and -O0
```
k2% test1.exe
-- TEST1: CPU SECONDS = ***** FLOP_RATE = 12.8796E+06 --
   52146.17913840042 52146.17913840042
```

Results for N=557 and -O3
```
k2% test1.exe
-- TEST1: CPU SECONDS = 0.435 FLOP_RATE = 79.5241E+07 --
   52146.17913840042 52146.17913840042
```

The -O3 optimization makes an enormous difference here. At the end of these notes you will find the listing, test1.w2f.f, of the transformed code that gets software pipelined. Both software pipelining and the optimizations used in test1.w2f.f are covered in the sections that follow. Note the drop in performance for the -O0 case in larger sized problem-mostly due to the higher number of L2 capacity misses.
INCREASE INSTRUCTION OVERLAP BY INNER LOOP UNROLLING

Loop unrolling reduces loop overhead (caused by branching, by reducing the number of needed loop iterations) and improves chances for instruction overlap. We can have vertical and horizontal unrolling, or a combination of both. Attention: in practice, inner loop unrolling should be attempted only if software pipelining (SWP) has not been successfull. For details, see the section on SWP.

```fortran
PROGRAM TEST2A
PARAMETER (N=2000000)
REAL*8 S1, S2, S3, S4, S5, S
REAL*8 AA(N), BB(N)
REAL*4 TM(2), TIME, OTIME, DTIME
C
DO I=1, N
   AA(I) = 1.00
   BB(I) = 1.001
END DO
TIME = DTIME(TM(2))
C
S = 0.00
DO I=1, N
   S = S + AA(I)*BB(I)
END DO
C
OTIME = DTIME(TM(2))
C
S1 = 0.00
S2 = 0.00
S3 = 0.00
S4 = 0.00
DO I=1,N, 4                          ! VERTICAL UNROLLING
   S1 = S1 + AA(I) * BB(I)
   S2 = S2 + AA(I+1) * BB(I+1)
   S3 = S3 + AA(I+2) * BB(I+2)
   S4 = S4 + AA(I+3) * BB(I+3)
END DO
C
S5 = S1 + S2 + S3 + S4
C
TIME=DTIME(TM(2))
C
WRITE(6,*) ' -- TEST2A: ROLLED LOOP: CPU SECONDS = ',OTIME,' --'
WRITE(6,*) ' -- TEST2A: UNROLLED LOOP : CPU SECONDS = ',TIME,' --'
WRITE(6,*) ' -- TEST2A: SPEED-UP = ',OTIME/TIME,' --'
WRITE(6,FMT='(" -- SUM: S5 =",E12.6,X,"S = ",E12.6," --")') S5,S
C
STOP
END
```

Texas A&M Computing & Information Services
INCREASE INSTRUCTION OVERLAP BY APPROPRIATELY GROUPING & UNROLLING TERMS

Floating-point addition and multiplication incur each 2 clock cycles. A madd \((a \times x + b)\) instruction costs 4 cycles. A madd improves performance by fetching and decoding one instruction instead of two. The gain from unrolling comes mostly from overlapping loads and stores of the next iteration \((i+1)\) with the adds, mul’s, and madds of the current one.

```c
PROGRAM TEST6A
      ILLUSTRATES WAYS TO PROMOTE INSTRUCTION OVERLAP & ESPECIALLY MADDS

INTEGER N
PARAMETER (N=500000)
REAL*8 XX(N), YY(N), A(N), B(N), C(N), D(N)
REAL*4 TM(2), TIME1, TIME2, OTIME, DTIME
      DO I=1, N
         XX(I) = 1.00
         A(I)  = 1.001
         B(I)  = 1.0002
         C(I)  = 1.0001
         D(I)  = 1.0003
      END DO

      TIME=DTIME(TM(2))
      DO I=1, N
         YY(I) = A(I)*XX(I)**3 + B(I)*XX(I)**2 + C(I)*XX(I) + D(I)
      END DO
      OTIME=DTIME(TM(2))
      DO I=1, N
         YY(I) = ( (A(I)*XX(I) + B(I)) * XX(I) + C(I) ) * XX(I) + D(I) +
               XX(I+1) + D(I+1)
      END DO
      TIME1=DTIME(TM(2))
      DO I=1, N, 2
         YY(I) = ( (A(I)*XX(I) + B(I)) * XX(I) + C(I) ) * XX(I) + D(I) +
               XX(I+1) + D(I+1)
         YY(I+1) = ( (A(I+1)*XX(I+1) + B(I+1)) * XX(I+1) + C(I+1) ) *
                     XX(I+1) + D(I+1)
      END DO
      TIME2=DTIME(TM(2))
      WRITE(6,*) '-- TEST6A: NO GROUPING OR UNROLLING - CPU SECONDS = ',
                     OTIME,' --'
      WRITE(6,*) '-- TEST6A: ONLY GROUPING ON         - CPU SECONDS = ',
                     TIME1,' --'
      WRITE(6,*) '-- TEST6A: GROUPING & UNROLLING ON  - CPU SECONDS = ',
                     TIME2,' --'
      WRITE(6,*) '-- TEST6A: SPEED-UP WITH GROUPING ONLY            = ',
                     OTIME/TIME1,' --'
      WRITE(6,*) '-- TEST6A: SPEED-UP WITH GROUPING & UNROLLING     = ',
                     OTIME/TIME2,' --'
      STOP
```

Texas A&M Computing & Information Services
END
k2% f77 -o test6a.exe -col120 test6a.f
k2% test6a.exe
-- TEST6A: NO GROUPING OR UNROLLING - CPU SECONDS = 0.4209600 --
-- TEST6A: ONLY GROUPING ON - CPU SECONDS = 0.3095780 --
-- TEST6A: GROUPING & UNROLLING ON - CPU SECONDS = 0.2960880 --
-- TEST6A: SPEED-UP WITH GROUPING ONLY = 1.359787 --
-- TEST6A: SPEED-UP WITH GROUPING & UNROLLING = 1.421739 --
k2%
SOFTWARE PIPELINING (SWP): THE CONCEPT

Critical for major performance improvement on R14000, software pipelining is a code optimization technique carried out by the compiler at the assembly language level. It affects only innermost loops. SWP strives to find a valid rearrangement of instructions (a schedule) in which instructions from different (successive) loop iterations are overlapped (in time), so as to engage concurrently as many pipelines as possible. One of the techniques that SWP uses to effect maximal instruction overlap is inner loop unrolling. We will illustrate the SWP concept through an example taken from M. Lam's (one of the main developers of the SWP method) tutorial notes [Lam92].

Suppose for the sake of simplicity that we have a machine that can simultaneously (same CP) issue only two instructions of which only one can reference memory, and, further, suppose that each access (loading a register from cache or storing a register to cache) takes one cycle, while an arithmetic operation (MUL, ADD) requires two cycles. One more assumption: all loads and stores are hits to the L1 cache because of small data requirements. In simple scalar fashion the following loop,

\[
\text{DO } \text{I}=1, \text{ N} \\
\quad \text{A(I)} = \text{A(I)*B + C} \\
\text{ENDDO}
\]

requires six cycles to execute any one iteration as the listing (for one iteration) below shows (loop overhead of branching and counter increment are ignored). Since scalars B and C are loop invariants, we pre-load them into registers, that is before loop iterations begin. They remain available throughout all iterations. Hence, no further accounting of their loading effect need be shown:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD</td>
<td>Read A(I)</td>
</tr>
<tr>
<td>2</td>
<td>MUL</td>
<td>Multiply A(I) by B</td>
</tr>
<tr>
<td>3</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ADD</td>
<td>Add C to A(I)*B</td>
</tr>
<tr>
<td>5</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>STORE</td>
<td>Write A(I) to Mem</td>
</tr>
</tbody>
</table>

With software pipelining, one possible way to schedule the above loop is as follows:
The above listing shows only four iterations of the software-pipelined code as a small part of the total sequence of N iterations. Note that while each iteration requires 8 cycles to flow through the "pipeline", four (overlapped) iterations require only 14 cycles to complete. The speedup obtained is 4*6/14 = 1.7 for four iterations. N iterations of the original loop require 6*N cycles to execute, whereas with this SWP, N iterations require 2*(N-1) + 8 cycles, because after the 8th cycle we obtain a result every 2 CPs. This results in a speedup of 6N/(2(N-1) + 8). For large N, this amounts to a speedup of 3. The block of instructions that repeats every 8-cycle period is called replication. Typical loops do not schedule so cleanly. They contain start up instructions called the windup portion of a schedule and an ending portion called winddown. SWP is normally very effective when the number of replications (the loop’s steady state) executed is significant.

The compiler does not SWP loops by default. To enable SWP use:

```
f90/f77/cc -R14000 -O3 ... files
```

To obtain an SWP status listing (file has .s suffix) use:

```
f90/f77/cc -R14000 -O3 -K -m -s -O3 -s -w -S -m -K -O3 -s -w -S
```
f90/f77/cc -R14000 -O3 prog.f -S

This generates the prog.s file. Use grep or egrep to look for messages starting with swps or swpf:

```
$     f90 -R14000 -O3 -o prog.exe prog.f -S
$     egrep -i 'swps|swpf' prog.s
```

Look out for messages that indicated aborted SWP as:

```
#:swpf> Loop line 48 wasn't pipelined because:
#:swpf> Branch out of the DoBody
#:swpf> Function call in the DoBody
```

**SWP CAN BE PREVENTED BY:**

- LONG LOOPS
- BRANCHING (E.G., SUB/FUNCTION CALLS, GOTOs)
- (SOMETIMES) CONDITIONALS (E.G., IF-THEN-ELSE)

**SWP LOOSES EFFICIENCY WHEN DATA DEPENDENCES OCCUR**

**SWP WORKS WELL WITH VECTORIZABLE LOOPS**

**SWP IS NOT EFFECTIVE WITH LOW (~5-10) ITERATION COUNTS**

**Attention**

You should always check whether a heavily used loop has been SWP’ed, and if not, for what reason. For an analysis of the SWP report see the next section.
SWP MESSAGES: An Example

```fortran
subroutine daxpy(n,a,x,y)
integer n
real*8 a, x(n), y(n)

do i=1, n
   y(i) = y(i) + a*x(i)
end do
return
end
```

To a major extent this is memory-bound (2 loads and one store per iteration) code. Hence, an optimization effort should focus in minimizing memory trips by efficient cache use and/or “hide” memory latencies by overlapping memory transfers with other operations.

```
k2% f77 -O3 -R14000 -S -c daxpy.f
k2% egrep -i swps daxpy.s
#<swps>
#<swps> Pipelined loop line 6 steady state
#<swps> 25 estimated iterations before pipelining
#<swps> 4 unrollings before pipelining
#<swps> 14 cycles per 4 iterations
#<swps> 8 flops (28% of peak) (madds count as 2)
#<swps> 4 flops (14% of peak) (madds count as 1)
#<swps> 4 madds (28% of peak)
#<swps> 12 mem refs (85% of peak)
#<swps> 3 integer ops (10% of peak)
#<swps> 19 instructions (33% of peak)
#<swps> 2 short trip threshold
#<swps> 7 ireg registers used.
#<swps> 15 fgr registers used.
```

This report tells us directly how fully and, therefore, how efficiently the R14K will be utilized under the schedule of instructions generated by SWP. The first important item to consider is that each replication block spanned 4 iterations that require 14 cycles to complete. Now, we can proceed to read with understanding the rest of the line messages if we bear in my mind the peak performance characteristics which we covered earlier: 2 flops/CP (1 from FPADD and 1 from FPMULT), 2 integer operations/CP (1 from ALU1 and 1 from ALU2), and 1 load or store per CP. So in 14 CPs the peak floating-point performance, when madds count as 2 floating-point ops, would be 28 flops. In 14 cycles, however, the above SWP schedule of instructions for the daxpy loop achieves only 8 floating-point ops, a ~28% (8/28) of peak performance. If a madd counts as one floating instruction, then the floating-point performance drops to 4 flops per 14 CPs, or ~14% (4/28) of peak floating-
point performance. This count intends to show how fully the floating-point capability of the R14K is being used in the above SWP schedule. And here that capability is not effectively exploited. Some times this is unavoidable because of the nature of the particular loop. However, floating-point performance, even at 100% utilization, is not, in general, an adequate metric for the overall CPU performance.

The above SWP schedule achieved (i.e., issued) 12 memory reference in 14 cycles. The peak is 14 such references (loads/stores). Hence, we achieved ~86% (12/14) of peak performance, fairly close to optimal.

On the integer ops side (that is, in regards to the utilization of ALU1 and ALU2), SWP achieved efficiencies of 3 integer ops per 14 cycles. Given the fact that in 14 cycles the maximum integer performance is 28 (2*14) ops, we obtained ~11% (3/28) of the peak integer performance here. Not good, but sometimes the nature of the code dictates no better.

The SWP report next says that within 14 cycles, the SWPed schedule managed to issue 19 instructions, while the peak is 56 (4*14), that is, a ~34% (19/56) of peak performance. This aspect of the performance, the number of concurrent instructions issued, is perhaps the most significant. Because it is the overlapped use of as many functional units as possible that will reduce total CPU time the most.

Next, the SWP message in the “2 short trip threshold” indicates that should the iteration count, n, be less than 2 (groups of four) iterations, a separate non-SWP’ed translation/compilation of the code will be generated. Otherwise read, small iteration counts are not good candidates for optimal performance through SWP.

Finally, the SWP message says that the each replication block uses 7 integer and 7 floating-point registers. We know that an R14K has 32 such registers of each type. So register availability is not a problem here. But in general, as in bigger loops with complicated expressions, the available number of registers can easily be exhausted. The compiler then may resort to using main memory locations for holding intermediate results, or find a way to reuse the existing registers better by adding additional code. In these situations loop splitting to two or more loops may remedy this problem.
INLINE TO HELP SWP

Example 1
Move the DO loop inside the subroutine.

This loop does not SWP:

```fortran
DO 10 I = 1,100
   CALL SUB (A(I), B(I), C(I))
10    CONTINUE
```

```fortran
SUBROUTINE SUB (X, Y, A)
   X = X + Y * Z ** 2
RETURN
END
```

This loop SWPs:

```fortran
CALL SUB (A, B, C)
```

```fortran
SUBROUTINE SUB (X, Y, Z)
   REAL X(100), Y(100), Z(100)
   DO 10 I = 1,100
      X(I) = X(I) + Y(I) * Z(I) ** 2
10    CONTINUE
RETURN
END
```
INLINE WITH THE STANDALONE INLINER

Use the standalone inliner to inline function calls

\texttt{f90/f77 \ldots -INLINE:must=sub1,sub2:list=ON \ldots file}

Here the standalone inliner looks in the input source file for sub1 and sub2 and inlines them. The list option lists to stderr inlining actions as they occur. Inlining has other useful suboptions such as \texttt{maxdepth}. For this and other inliner options see the ipa man page.

When subroutines sub1, sub2, ... are not part of the input source, but instead they are in separate files sub1.f and sub2.f respectively, then one must follow a different procedure for inlining sub1 and sub2. Assume, say, that subroutines sub1 and sub2 are called within the main program, main.f. then:

\texttt{f90/f77 \ldots -IPA -c sub1.f}

\texttt{f90/f77 \ldots -IPA -c sub2.f}

\texttt{f90/f77 \ldots -INLINE:must=sub1_:file=sub1.o \ }
\hspace{1cm} -INLINE:must=sub2_:file=sub2.o main.f

Caution! Inlining increases the size of your code. The more extensive the inlining, the larger the increase. Inlining must be carried out carefully and selectively if it is to prove effective. Generally, one inlines only frequently called subroutines and functions which do not consume very much time. Global inlining is rarely of much benefit in sizable codes.

For more information on inlining consult the \texttt{ipa} man page.

\textbf{Note}

Inlining can be of great value in some codes. This occurs in codes where a few subroutines, called many times within loops, dominate the execution profile of a program. The tracking of such routines is done using the performance monitoring utilities \texttt{prof} and \texttt{ssrun}. Such tracking should be done with executables generated with optimization options that DO NOT include inlining (e.g., -O0, -O1).
SPLIT WIDE LOOPS TO HELP ENABLE SWP

DO  i=N1, NMAX
   x11(i)=x7(i)-x1(i)
   x28(i)=x8(i)-x2(i)
   x35(i)=x5(i)-x3(i)
   z35(i)=z5(i)-z3(i)
   z46(i)=z6(i)-z4(i)
   aj1(i)=x11(i)+x28(i)-x35(i)-x46(i)
   aj1(i)=y17(i)+y28(i)-y35(i)-y46(i)
   aj1(i)=z17(i)+z28(i)-z35(i)-z46(i)

   ...

   ...

   ...

   ...

   ...

   ...

   ...

   ...

   ...

   a17(i)=x17(i)+x46(i)
   a28(i)=x28(i)+x35(i)
   aj7(i)=a17(i)-a28(i)
   aj8(i)=b17(i)-b28(i)
   aj9(i)=c17(i)-c28(i)
ENDDO

grep swpf foo.s:
#<swpf> Loop line 44 wasn’t pipelined -- couldn’t allocate enough ireg registers.
#<swpf>
The above loop should be split into two or more loops to enhance prospects for SWP.
FUSE THIN LOOPS TO IMPROVE SWP

On the R14000 upto four instructions can be issued on the same cycle to any of the five independent execution units. SWP’s role is to keep concurrently and productively busy as many of these as possible. Hence, for Do-loops involving a limited variety of operations making efficient use of the machine is difficult. For example, loops such as the three shown below by themselves engage only two or three execution units.

```plaintext
DO I=1, N
   A(I) = B(I+K)
END DO

DO I=1, N
   C(I) = B(I) + X
END DO

DO I=1, N
   D1 = D1 * E(I)
END DO
```

All three loops involve the load/store unit and the ALU1 (for integer arithmetic needed in incrementing the loop index). The second and third loops, in addition, involve FPADD and FPMULT respectively. On the other hand, the fused loop below provides a richer variety of instructions to schedule efficiently with SWP because it can concurrently engage four execution units.

```plaintext
DO I=1, N
   A(I) = B(I+K)
   C(I) = B(I) + X
   D1 = D1 * E(I)
END DO
```

Loop fusing is often beneficial in another respect, efficient cache reuse. More on this in a later section.

**Caution**

Loop fusion is not always permissible. It is not permissible when it introduces or violates data dependences.
OUTER LOOP UNROLLING: GENERAL CONCEPT

The principal objective of outer loop unrolling is to promote instruction overlap by providing a larger number of executable instructions within each iteration of the innermost loop. It also boosts the floating-point operations to memory accesses ratio. That is, it enhances cache and register reuse.

S(I,J) below is any executable statement that depends on I and J

```
DO I=1, N
   DO J=1, N
      S(I, J)
   END DO
END DO
```

with 4-way outer-loop unrolling becomes

```
DO I=1, N-3, 4
   DO J=1, N
      S(I, J)
      S(I+1, J)
      S(I+2, J)
      S(I+3, J)
   END DO
END DO
DO I=I, N
   DO J=1, N
      S(I, J)
   END DO
END DO
```

Note that the initial value of I in the last loop is either greater than N, in which case the last loop does not execute, or the initial value of I for this loop is N-3.

Caution

Outer loop unrolling is not always permissible. It is not permissible when it violates the data dependence relations that existed before the unrolling.
OUTER LOOP UNROLLING FOR LOWER MEM_REFS TO FLOP RATIO & HIGHER INSTRUCTION OVERLAP

The performance results obtained here are meant to be compared with those of the matrix multiplication example, test1.f, on page 25.

```fortran
PROGRAM TEST2BA
PARAMETER (N=257, NSQ=N*N)
REAL*8 AA(N,N), BB(N,N), CC(N,N)
REAL*4 TM(2), TIME1, TIME2, FLOP_RATE, DTIME
DATA AA/NSQ*1.1001/, BB/NSQ*2.5012/, CC/NSQ*0.00/

C
TIME1=DTIME(TM(2))
C
DO I=1, N-1, 2                   ! Two-level OUTER LOOP UNROLLING
  DO J=1, N-1, 2                ! HIGHER INSTRUCTION OVERLAP
    DO K=1, N
      CC(I,J) = CC(I,J) + AA(I,K) * BB(K,J)
      CC(I,J+1) = CC(I,J+1) + AA(I,K) * BB(K,J+1)
      CC(I+1,J) = CC(I+1,J) + AA(I+1,K) * BB(K,J)
      CC(I+1,J+1) = CC(I+1,J+1) + AA(I+1,K) * BB(K,J+1)
    END DO
  END DO
  DO J=J, N
    DO K=1, N
      CC(I,J) = CC(I,J) + AA(I,K) * BB(K,J)
      CC(I+1,J) = CC(I+1,J) + AA(I+1,K) * BB(K,J)
    END DO
  END DO
END DO

C
DO I=I, N
  DO J=1, N-1, 2
    DO K=1, N
      CC(I,J) = CC(I,J) + AA(I,K) * BB(K,J)
      CC(I,J+1) = CC(I,J+1) + AA(I,K) * BB(K,J+1)
    END DO
  END DO
  DO J=J, N
    DO K=1, N
      CC(I,J) = CC(I,J) + AA(I,K) * BB(K,J)
    END DO
  END DO
END DO

TIME2=DTIME(TM(2))
FLOP_RATE = (2*N*N*N)/TIME2
C
WRITE(6,FMT=’("-- TEST2BA: CPU SECONDS = ", F9.7," FLOP_RATE = ", + 3PE10.4," --")’) TIME2, FLOP_RATE
C
WRITE(6, FMT=’(2(E12.6,X))’) CC(102,13), CC(156,13)
STOP
END

k2% f77 -R14000 -col120 -o test2ba.exe test2ba.f
```

Texas A&M Computing & Information Services
k2% test2ba.exe
-- TEST2BA: CPU SECONDS = 2.2874939  FLOP_RATE = 148.41E+05 --
0.707154E+03 0.707154E+03

k2% f77 -R14000 -col120 -O3 -o test2ba.exe test2ba.f
k2% test2ba.exe
-- TEST2BA: CPU SECONDS = 0.1632310  FLOP_RATE = 207.98E+06 --
0.707154E+03 0.707154E+03

Note that a speedup was realised in both cases: with and without -O3. The valid comparisons here are with results obtained in test1.f. The corresponding speedups are 1.23 and 1.10.
OUTER LOOP UNROLLING WITH PREFETCHING FOR MAX. REGISTER REUSE & INSTRUCTION OVERLAP

PROGRAM TEST2BB
PARAMETER (N=257, NSQ=N*N)
REAL*8 AA(N,N), BB(N,N), CC(N,N)
REAL*8 DD1, DD2, DD3, DD4, DD5, DD6, DD7, DD8
REAL*4 TM(2), TIME, FLOP_RATE, DTIME
DATA AA/NSQ*1.0001/, BB/NSQ*1.0012/, CC/NSQ*0.00/
C
TIME=DTIME(TM(2))
C
DO I=1, N-1, 2               ! OUTER LOOP UNROLLING FOR
  DO J=1, N-1, 2              ! MAXIMUM REGISTER REUSE
    DD1 = CC(I,J)
    DD2 = CC(I,J+1)
    DD3 = CC(I+1,J)
    DD4 = CC(I+1,J+1)
    DO K=1, N
      DD1 = DD1 + AA(I,K) * BB(K,J)
      DD2 = DD2 + AA(I,K) * BB(K,J+1)
      DD3 = DD3 + AA(I+1,K) * BB(K,J)
      DD4 = DD4 + AA(I+1,K) * BB(K,J+1)
    END DO
    CC(I,J) = DD1
    CC(I,J+1) = DD2
    CC(I+1,J) = DD3
    CC(I+1,J+1) = DD4
  END DO
  DO J=J, N
    DD1 = CC(I,J)
    DD3 = CC(I+1,J)
    DO K=1, N
      DD1 = DD1 + AA(I,K) * BB(K,J)
      DD3 = DD3 + AA(I+1,K) * BB(K,J)
    END DO
    CC(I,J) = DD1
    CC(I+1,J) = DD3
  END DO
END DO
C
DO I=I, N
  DO J=1, N-1, 2
    DD1 = CC(I,J)
    DD2 = CC(I,J+1)
    DO K=1, N
      DD1 = DD1 + AA(I,K) * BB(K,J)
      DD2 = DD2 + AA(I,K) * BB(K,J+1)
    END DO
    CC(I,J) = DD1
    CC(I,J+1) = DD2
  END DO
END DO
DO J=J, N
   DD1 = CC(I,J)
   DO K=1, N
      DD1 = DD1 + AA(I,K) * BB(K,J)
   END DO
   CC(I,J) = DD1
END DO
END DO
TIME=DTIME(TM(2))

C
FLOP_RATE = (2*N*N*N)/TIME
WRITE(6,FMT='("-- TEST2BB: CPU SECONDS = ", F5.3,
+ " FLOP_RATE = ",3PE10.4," --")') TIME, FLOP_RATE
WRITE(6, FMT='(2(E12.6,X))') CC(102,13), CC(134,156)
STOP
END

k2% f77 -R14000 -col120 -o test2bb.exe test2bb.f
k2% test2bb.exe
-- TEST2BB: CPU SECONDS = 1.429  FLOP_RATE = 237.50E+05 --
0.257334E+03 0.257334E+03

k2% f77 -R14000 -col120 -O3 -o test2bb.exe test2bb.f
k2% test2bb.exe
-- TEST2BB: CPU SECONDS = 0.785  FLOP_RATE = 432.66E+05 --
0.257334E+03 0.257334E+030/

Note here the destructive interaction between the actions of -O3 and prefetching with outer unrolling.
INTERCHANGE LOOPS FOR BETTER CACHE USE

PROGRAM TEST1A1
C The PRINT statement at the end forces f90 to generate an executable.
PARAMETER (N=257, NSQ=N*N)
REAL*8 CC(N,N), AA(N,N), BB(N,N), DD(N,N)
REAL*8 TIME1, TIME2, TIME3, FLOP_RATE1, FLOP_RATE2
C
DATA AA/NSQ*8.099/, BB/NSQ*11.638/, CC/NSQ*0.00/
C
CALL CPU_TIME(TIME1)
C
DO K=1,N ! Matrix Multiplication
  DO I=1, N
    DO J=1, N
      CC(I,J) = CC(I,J) + AA(I,K)*BB(K,J)
    END DO
  END DO
END DO
C
CALL CPU_TIME(TIME2)
C
DO K=1,N                       ! Exchange the J & I loops
  DO J=1, N
    DO I=1, N
      CC(I,J) = CC(I,J) + AA(I,K)*BB(K,J)
    END DO
  END DO
END DO
C
CALL CPU_TIME(TIME3)
C
FLOP_RATE1 = (2*N*N*N)/(TIME2-TIME1)
FLOP_RATE2 = (2*N*N*N)/(TIME3-TIME2)
PRINT *, '-- TEST1A1: J-LOOP INNERMOST: CPU SECONDS = ', TIME2-TIME1,
  ' FLOP_RATE = ', FLOP_RATE1
PRINT *, '-- TEST1A1: I-LOOP INNERMOST: CPU SECONDS = ', TIME3-TIME2,
  ' FLOP_RATE = ', FLOP_RATE2
PRINT *, '-- TEST1A1: SPEEDUP = ', (TIME2-TIME1)/(TIME3-TIME2)
STOP
END

k2% f90 -extend_source -o test1a1.exe test1a1.f
k2% test1a1.exe
-- TEST1A1: J-LOOP INNERMOST: CPU SECONDS =  3.47 FLOP_RATE =  9776420.19
-- TEST1A1: I-LOOP INNERMOST: CPU SECONDS =  2.91 FLOP_RATE = 11674029.22
-- TEST1A1: SPEEDUP =  1.19
2*48447.665193706554

Results for N=557
-- TEST1A1: J-LOOP INNERMOST: CPU SECONDS = 132.41 FLOP_RATE = 2610022.73
-- TEST1A1: I-LOOP INNERMOST: CPU SECONDS =  69.86 FLOP_RATE = 4947019.68
-- TEST1A1: SPEEDUP =  1.90
2*105001.35997235242
k2%

Comments
The second loop nest has superior locality since, now, both CC and AA are
accessed sequentially.
**FUSE OR SPLIT LOOPS FOR BETTER CACHE USE**

The sample code below illustrates the case where statements that use the same data variables should be moved together as much as possible for better cache reuse. This may not be the case, however, if such statements are long, complicated and involve large arrays, such that capacity misses occur with every iteration. In fact, for such cases loop splitting should be seriously considered.

```fortran
PROGRAM TEST2AC
PARAMETER (N=2000000)
REAL*8 A1(N), A2(N), BB(N)
REAL*4 TM(2), TIME, OTIME, DTIME
C
DATA A1/N*0.0000001/, A2/N*0.0000001/, BB/N*0.0000000002/
TIME = DTIME(TM(2))
C
DO I=1, N
   A1(I) = A1(I) + (mod(N,2) + 1.0) * BB(I)
END DO
XA1 = 0.00
DO I=1, N
   XA1 = XA1 + A1(I)*A1(I)
END DO
C
OTIME = DTIME(TM(2))
C
XA2 = 0.00
DO I=1, N ! Fused Loops
   A2(I) = A2(I) + (mod(N,2) + 1.0) * BB(I)
   XA2 = XA2 + A2(I)*A2(I)
END DO
C
TIME = DTIME(TM(2))
C
WRITE(6,*) ' -- TEST2AC: UNFUSED LOOP: CPU SECONDS =', OTIME, ' --'
WRITE(6,*) ' -- TEST2AC: FUSED LOOP : CPU SECONDS =', TIME, ' --'
WRITE(6,*) ' -- TEST2AC: SPEED-UP =', OTIME/TIME, ' --'
C
WRITE(6,FMT=(' " -- SUMS: XA1 =''E12.6," XA2 =''E12.6," --"')) XA1, XA2
STOP
END
```

In the last loop any cache line involved in referencing A2 and BB is transferred just once from main memory. Whereas in the two loops above the last one, each involved cache line is accessed twice for the A1 array and only once for BB.
SOME IN-CODE OPTIMIZATION DIRECTIVES

In Fortran

C*$* NOCONCURRENTIZE
C*$* ROUNDOFF (0-3)
C*$* SCALAR OPTIMIZE (0-3)
C*$* UNROLL ("n")
C*$* [NO]INLINE (sub1,sub2,...) [HERE | ROUTINE | GLOBAL ]

In C

#pragma [no] concurrentize
#pragma scalar
#pragma unroll (n)
#pragma [no]inline [here | routine | global ] (fn1, fn2, ...)

Example

PROGRAM TEST1D                               ! MATRIX MULTIPLY
PARAMETER (N=157, NSQ=N*N)
REAL*8 AA(N,N), BB(N,N), CC(N,N)
REAL*4 TM(2), TIME, OTIME, DTIME
DATA AA/NSQ*1.1001/, BB/NSQ*2.5012/, CC/NSQ*0.00/
C
TIME=DTIME(TM(2))
C
C*$*NOCONCURRENTIZE
C*$*SCALAR OPTIMIZE (0)                        ! DISABLE SCALAR OPTIMIZATION
C
DO I=1,N                                     ! HIGH SCHOOL MATRIX MULTIPLY
   DO J=1, N
      DO K=1, N
         CC(I,J) = CC(I,J) + AA(I,K)*BB(K,J)
      END DO
   END DO
END DO
C
OTIME=DTIME(TM(2))
C
C*$*SCALAR OPTIMIZE (3)
C*$*ROUNDOFF (3)
DO I=1,N                                     ! HIGH SCHOOL MATRIX MULTIPLY
   DO J=1, N
      DO K=1, N
         CC(I,J) = CC(I,J) + AA(I,K)*BB(K,J)
      END DO
   END DO

Texas A&M Computing & Information Services
END DO
END DO

C

TIME=DTIME(TM(2))
WRITE(6,*)
+ '-- TEST1D: SCALAR OPTIMIZATION OFF: CPU SECONDS = ',OTIME,' --'
WRITE(6,*)
+ '-- TEST1D: SCALAR OPTIMIZATION ON : CPU SECONDS = ',TIME,' --'
WRITE(6,*)
+ '-- TEST1D: SPEED-UP
+ OTIME/TIME,' --'
C

PRINT *, CC(97,56), CC(34,65)
STOP
END

k2% /usr/lib/pfa test1d.f
k2% f77 -R14000 -o test1d.exe test1d.m
k2% test1d.exe
-- TEST1D: SCALAR OPTIMIZATION OFF: CPU SECONDS = 0.8194560 --
-- TEST1D: SCALAR OPTIMIZATION ON : CPU SECONDS = 0.3822630 --
-- TEST1D: SPEED-UP = 2.143697 --
863.9930176799991 863.9930176799991

NOTE: Use of the -O3 option in conjunction with some other scalar options does not yield the best performance results. Mixing -O3 with other options or techniques should be done carefully and stepwise.

References
For a fuller account on C and C++ pragmas please consult the MIPSpro C and
C++ Pragmas reference of the “SGI Developer” set via the insight utility. For For-
tran compiler directives consult the MIPSpro 7 Fortran 90 Commands and
Directives, also an SGI Developer document.
BLOCKING/TILING: GENERAL CONCEPT

Blocking is carried out to improve data locality of reference in nested loops where the data to be processed by the innermost do-loop do not fit in the available cache space. Its general idea is to limit the range of the memory reference sweep involved in the inner do-loop and add, at the same time, appropriate nesting levels over the original nest structure. This way fewer cache misses are encountered, while at the same time doing more calculations with the data already in the cache. Thus, the original execution sequence is altered so that outer loop iterations are executed more before inner loop iterations are completed.

Below, suppose $S(I,J)$ is any executable statement in the $i$th and $j$th iteration, and where the unfavorable memory reference pattern occurs along the $I$th index, then

\[
\begin{align*}
\text{DO } & J=1, \ N \\
\text{DO } & I=1, \ N \\
& S(I,J) \\
\text{END DO} \\
\text{END DO}
\end{align*}
\]

transforms to:

\[
\begin{align*}
& \text{INTEGER WSZ} \\
& \text{DO } M=1, \ N, \ WSZ \\
& \text{DO } J=1, \ N \\
& \text{DO } I=M, \ \min(M+\text{WSZ}-1, N) \\
& \hspace{1cm} S(I,J) \\
& \text{END DO} \\
& \text{END DO} \\
& \text{END DO}
\end{align*}
\]

Above, the $N$ iterations of the $I$-loop are split into chunks of $\text{WSZ}$ (plus a remainder that’s taken care of by the $\text{MIN}$ function). It is these $\text{WSZ}$ iterations that the inner loop executes, and it is during those iterations that we want, in the general case, to do more computation with the data available in the cache. In this case, the additional computation with the data already in the cache is provided by the $J$-loop. It is important that the $M$-loop envelop both the $I$-loop and the $J$-loop. If it were to to wrap around the $I$-loop only there would be no benefit. In fact, the resultant loop in that case would be a bit slower.
PROGRAM BTEST1
C
C Measure efficacy of "tiling" to promote locality of reference and
C data reuse in the L2 cache. Use the R14K's hardware performance
C counters with perfex to monitor L2 cache misses and CPU cycles.
C
C Compile with: f77 -o btest1.exe -R14000 -col120 btest1.f -lperfex
C
INTEGER BLK_SZ, NSZ, NSQ
PARAMETER (NSZ=513, NSQ=NSZ*NSZ, BLK_SZ=128)
REAL*8 AA(NSZ,NSZ), BB(NSZ,NSZ)
REAL*8 CC1(NSZ,NSZ), CC2(NSZ,NSZ), CC3(NSZ,NSZ), CC4(NSZ,NSZ)
INTEGER*8 C0, C1 ! EVENT COUNTERS
INTEGER*4 E0, E1 ! EVENTS TO MONITOR
C
E0 = 0 ! CPU CYCLES EVENT IN COUNTER C0
E1 = 26 ! L2 CACHE MISSES EVENT IN COUNTER C1
C
DO I=1, NSQ ! INITIALIZE DATA
   AA(I,1) = RAND() * 10.00
   BB(I,1) = SIN(FLOAT(I))
   CC1(I,1) = 0.00
   CC2(I,1) = 0.00
   CC3(I,1) = 0.00
   CC4(I,1) = 0.00
END DO
C
ISTAT = START_COUNTERS( E0, E1 ) ! MATRIX MULTIPLY WITH POOREST
ISTAT = READ_COUNTERS( E0, C0, E1, C1 ) ! MEMORY ACCESS PATTERN FOR CC1 & BB
ISTAT = PRINT_COUNTERS( E0, C0, E1, C1 )
C
ISTAT = START_COUNTERS( E0, E1 ) ! MATRIX MULTIPLY WITH OUTER
ISTAT = READ_COUNTERS( E0, C0, E1, C1 )
WRITE(6, FMT='(//,5X,"***** FIRST LOOP: NO TILING ******")') BLK_SZ
CALL FLUSH(6)
ISTAT = PRINT_COUNTERS( E0, C0, E1, C1 )
C
ISTAT = START_COUNTERS( E0, E1 )
ISTAT = READ_COUNTERS( E0, C0, E1, C1 )
WRITE(6, FMT='(//,5X,"***** SECOND LOOP: 1-LEVEL OUTER TILING
BLK_SZ=",I3," ******")') BLK_SZ
CALL FLUSH(6)
ISTAT = PRINT_COUNTERS( E0, C0, E1, C1 )

Texas A&M Computing & Information Services
C
C  ! MATRIX MULTIPLY WITH OUTER
DO L = 1, NSZ, BLK_SZ
  DO M = 1, NSZ, BLK_SZ
    DO I = 1, NSZ
      DO K=L, MIN(L+BLK_SZ-1,NSZ)
        DO J=M, MIN(M+BLK_SZ-1,NSZ)
          CC3(I,J) = CC3(I,J) + AA(I,K)*BB(K,J)
        END DO
      END DO
    END DO
  END DO
END DO
C
CISTAT = READ_COUNTERS( E0, C0, E1, C1 )
WRITE(6, FMT='(//,5X,"***** THIRD LOOP: 2-LEVEL OUTER TILING
BLK_SZ="",I3," *****")' ) BLK_SZ
CALL FLUSH(6)
CISTAT = PRINT_COUNTERS( E0, C0, E1, C1 )
C
CISTAT = START_COUNTERS( E0, E1 )
C
DO N = 1, NSZ, BLK_SZ
  ! OUTER 3-LEVEL TILING
  DO L = 1, NSZ, BLK_SZ
    DO M = 1, NSZ, BLK_SZ
      DO I = N, MIN(N+BLK_SZ-1,NSZ)
        DO K=L, MIN(L+BLK_SZ-1,NSZ)
          DO J=M, MIN(M+BLK_SZ-1,NSZ)
            CC4(I,J) = CC4(I,J) + AA(I,K)*BB(K,J)
          END DO
        END DO
      END DO
    END DO
  END DO
C
CISTAT = READ_COUNTERS( E0, C0, E1, C1 )
WRITE(6, FMT='(//,5X,"***** FOURTH LOOP: 3-LEVEL OUTER TILING
BLK_SZ="",I3," *****")' ) BLK_SZ
CALL FLUSH(6)
CISTAT = PRINT_COUNTERS( E0, C0, E1, C1 )
C
DO I=1, NSQ!
  VERIFY CORRECTNESS
  IF ( ABS(CC1(I,1) - CC2(I,1)) .GT. 1.0E-9 .OR.
    + ABS(CC1(I,1) - CC3(I,1)) .GT. 1.0E-9 .OR.
    + ABS(CC1(I,1) - CC4(I,1)) .GT. 1.0E-9 ) THEN
    WRITE(6,*) '****** ERROR ENCOUNTERED ******'
    WRITE(6, FMT='("CC1("I6",",1)="E12.8,2X,"CC2("I6",",1)="E12.8,2X,
                       + "CC3("I6",",1)="E12.8,2X,"CC4("I6",",1)="E12.8")
    + I,CC1(I,1), I,CC2(I,1), I,CC3(I,1), I,CC4(I,1)
  END IF
STOP
END DO
STOP
END

k2% f77 -o btest1.exe -Rl4000 -col120 btest1.f -lperfex
k2% btest1.exe

******* FIRST LOOP: NO TILING ********
Cycles ........................................................   7715668219
<table>
<thead>
<tr>
<th>Loop Description</th>
<th>Cycles</th>
<th>Secondary data cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECOND LOOP: 1-LEVEL OUTER TILING BLK_SZ=128</td>
<td>49,275,975,86</td>
<td>88,173,5</td>
</tr>
<tr>
<td>THIRD LOOP: 2-LEVEL OUTER TILING BLK_SZ=128</td>
<td>48,763,138,22</td>
<td>39,960,3</td>
</tr>
<tr>
<td>FOURTH LOOP: 3-LEVEL OUTER TILING BLK_SZ=128</td>
<td>48,704,647,09</td>
<td>40,239,7</td>
</tr>
</tbody>
</table>
**BLOCKING & EFFICIENT CACHE USE: EXAMPLE 2**

```fortran
PROGRAM TEST1B
INTEGER BLK_SZ, N
PARAMETER (BLK_SZ=16, N=1024)
REAL*8 AA(N,N)
REAL*4 TM(2), TIME, OTIME, DTIME

C
PI = 4.0 * ATAN(1.0)
ANG_INC = PI/N
DO J=1,N
   DO I=1, N
      AA(I,J) = 0.0
   END DO
END DO

! Initialize bottom & left boundaries
DO J=1, N
   AA(1,J) = 10.0 * SIN( J*ANG_INC )
   AA(J,1) = AA(1,J)
END DO
TIME=DTIME(TM(2))

DO I=2, N-1
   DO J=2, N-1
      AA(I,J) = ( AA(I-1,J)+AA(I,J-1)-AA(I+1,J)+AA(I,J+1) )*0.5
   END DO
END DO

OTIME=DTIME(TM(2))

DO L=2, N-1, BLK_SZ
   DO M=2, N-1, BLK_SZ
      DO I=L, MIN0(L+BLK_SZ-1,N-1)
         DO J=M, MIN0(M+BLK_SZ-1,N-1)
            AA(I,J) = ( AA(I-1,J)+AA(I,J-1)-AA(I+1,J)+AA(I,J+1) )*0.5
         END DO
      END DO
   END DO
END DO

TIME=DTIME(TM(2))
WRITE(6,*) '-- TEST1: UNBLOCKED LOOP: CPU SECONDS = ',OTIME,' --'
WRITE(6,*) '-- TEST1: BLOCKED LOOP  : CPU SECONDS = ',TIME,' --'
WRITE(6,*) '-- TEST1: SPEED-UP                    = ',OTIME/TIME,' --'
STOP
END
```

```
% f77 -o test1b.exe -col120 test1b.f
% date
Sat Feb 17 14:43:08 CST 2001
% test1b.exe
-- TEST1: UNBLOCKED LOOP: CPU SECONDS = 8.394798 --
-- TEST1: BLOCKED LOOP  : CPU SECONDS = 1.018178 --
-- TEST1: SPEED-UP                    = 8.244923 --

For N=2048 and BLK_SZ=32
% test1b.exe
-- TEST1: UNBLOCKED LOOP: CPU SECONDS = 11.59967 --
-- TEST1: BLOCKED LOOP  : CPU SECONDS = 2.230888 --
-- TEST1: SPEED-UP                    = 5.199574 --

For N=2049 and BLK_SZ=128
% test1b.exe
-- TEST1: UNBLOCKED LOOP: CPU SECONDS = 3.945392 --
-- TEST1: BLOCKED LOOP  : CPU SECONDS = 2.742263 --
-- TEST1: SPEED-UP                    = 1.438736 --
```
/* C-Version of PROGRAM TEST1B; Link: -lm */

#include <stdio.h>
#include <time.h>
#include <math.h>
#define blk_sz 128
#define n 1024
#define min(A,B) ((A) < (B) ? (A) : (B))

void main()
{
    int i, j, l, m;
    double aa[n][n], pi, ang_inc;
    clock_t time1, time2, time3;
    pi = 4.0 * atan(1.0);
    ang_inc = pi/n;

    for( i=0; i<n; i++ )
        for( j=0; j<n; j++ )
            aa[i][j] = 0.0;

    for( j=0; j<n; j++ ) /* Initialize bottom and left extremities */
    {
        aa[0][j] = 10.0 * sin( j*ang_inc );
        aa[j][0] = aa[0][j];
    }

    time1 = clock();

    for( j=1; j<n-1; j++ )
        for( i=1; i<n-1; i++ )
            aa[i][j] = ( aa[i-1][j]+aa[i][j-1]-aa[i+1][j]+aa[i][j+1] ) * 0.5;

    time2 = clock();

    for( m=1; m<n-1; m+=blk_sz )
        for( l=1; l<n-1; l+=blk_sz )
            for( j=m; j < min(m+blk_sz, n-1); j++ )
                for( i=l; i < min(l+blk_sz, n-1); i++ )
                    aa[i][j] = ( aa[i-1][j]+aa[i][j-1]-aa[i+1][j]+aa[i][j+1] ) * 0.5;

    time3 = clock();

    printf( "-- TEST1: UNBLOCKED LOOP: CPU SECONDS = %12.9f
", (double)(time2-time1)*(1E-6) );
    printf( "-- TEST1: BLOCKED LOOP: CPU SECONDS = %12.9f
", (double)(time3-time2)*(1E-6) );

    printf( "-- TEST1: SPEED UP = %12.9f
", (double)(time2-time1)/(time3-time2) );

    printf( "aa[999][999] = %12.9f aa[601][712] = %12.9f
", aa[999][999], aa[601][712] );
}

Comment

Please note that the above is not an exact translation of its Fortran counterpart in the previous page. It is, however, an equivalent code and, especially, in the sense that the memory access patterns are in each case, Fortran and C, equally unfavorable.
ADDITIONAL f90/f77/cc DIRECTIVES and FLAGS

C*$ BLOCKING SIZE (n1,n2)
C*$ BLOCKING SIZE (n1)
C*$ BLOCKING SIZE (,n2)
C*$ BLOCKABLE (I,J [,K ...] )
C*$ NO BLOCKING
C*$ INTERCHANGE (I, J [,K ...]
C*$ FUSE [(n [,level] )]
C*$ NO FUSION
C*$ FISSION [(n)]
C*$ NO FISSION

For more information on these and other options consult the f77 man page. All of the above directives are also available in C. In C code use the #pragma specifier instead of C*S*.

-OPT:IEEE_arithmetic=3

Specifies the level of conformance to the IEEE 754 floating-point standard in regard to roundoff and overflow behaviour. At level 1 (the default), no optimizations are carried out which might produce less accurate results than those of strict adherence to the IEEE 754. Level 2 allows the use of operations which may produce less accurate inexact results (but accurate exact results) on the target hardware. Examples are the recip (reciprocal) and rsqrt (reciprocal square root) primitive operations. Level 3 allows arbitrary but mathematically valid transformations, even if they may produce inaccurate results vis-as-vis the IEEE 754 specified operations, or may overflow or underflow for a valid operand range. An example is the replacement of x/y with x*recip(y).

-OPT:roundoff=3

Specifies the level of acceptable departure from source language floating-point roundoff and overflow semantics. Level 0 (the default for -O0 to -O2), does no opti-
imizations which might affect the floating-point behaviour. Level 1, allows simple transformations which might cause limited roundoff or overflow differences. Level 2 (the default for -O3) allows more extensive transformations, such as the execution of iterations in a reduction loop in different order. Level 3 enables any mathematically valid transformation. Level 3, in general, yields the best SWP performance.

**Additional optimization options**

- LNO: . . .
- IPA: . . .
- OPT: . . .

For specific information on these see their man pages: lno, ipa, opt.

**Recommended compilation options for stable code**

f90/f77/cc -R14000 -O3 -OPT:=IEEE_arithmetic=3:roundoff=3 -lfastm

We emphasize that the above combination of options be carried only for numerically stable and robust code. A careful approach to using high level optimizations should involve compiling separately those routines that are heavily used and are known to be robust. The need for numerically stable code is dictated by the fact, among other, that the -OPT:roundoff=3 option can cause arithmetic operations to be commuted and re-associated, thereby providing a source for numerical differences with the un-optimized version of the code.

**References**

For a fuller account on C and C++ pragmas please consult the MIPSpro C and C++ Pragmas reference of the “SGI Developer” set via the insight utility. For Fortran compiler directives consult the MIPSpro 7 Fortran 90 Commands and Directives, also an SGI Developer document. Please note that the C/C++ implementation may have some directives missing or different to the Fortran version.
MINIMIZE USE OF FLOATING-POINT DIVISION

Division is an expensive (~19 CPs) operation. Worse, it is not pipelined.

This loop is slower:

```
DO 10 I = 1,N
   A(I) = B(I)/4.
10    CONTINUE
```

This loop is more efficient:

```
DO 10 I = 1,N
   A(I) = B(I) * .25
10    CONTINUE
```
TRANFORMED SOURCE & SWP LISTINGS

k2% f77 -flist -R14000 -O3 -o test1.exe test1.f

/usr/lib32/cmplrs/be translates /tmp/ctmB.BAaa004qP into test1.w2f.f, based on source test1.f

k2% cat test1.w2f.f

C ***********************************************************
C Fortran file translated from WHIRL Sat Sep 6 14:18:45 1997
C ***********************************************************
PROGRAM MAIN
IMPLICIT NONE

C       **** Variables and functions ****
C
REAL*8 AA(257_8, 257_8)
SAVE AA
REAL*8 BB(257_8, 257_8)
SAVE BB
REAL*8 CC(269_8, 257_8)
REAL*4 TM(2_8)
REAL*4 TIME
INTEGER*4 I
INTEGER*4 J
INTEGER*4 K
REAL*4 FLOP_RATE
EXTERNAL dtime
REAL*4 dtime

C       **** Temporary variables ****
C
REAL*4 tmp0
INTEGER*4 tile2I
INTEGER*4 tile2J
INTEGER*4 tile1K
INTEGER*4 tile1I
REAL*8 mi0
REAL*8 mi1
REAL*8 mi2
REAL*8 mi3
REAL*8 mi4
REAL*8 mi5
REAL*8 mi6
REAL*8 mi7
INTEGER*4 wd_K0
REAL*8 mi8
REAL*8 mi9
INTEGER*4 I0
INTEGER*4 wd_J
INTEGER*4 K0
REAL*8 mi10
REAL*8 mi11
REAL*8 mi12
REAL*8 mi13
INTEGER*4 I1
INTEGER*4 wd_K
REAL*8 mi14
INTEGER*4 I2
REAL*4 preg

C       **** Initializers ****
C
DATA AA / 66049 * 8.099000000000002D00 /
**Scalar Code Optimization I**

```
DATA BB / 66049 * 1.1638D+01 /

*** statements ***

tmp0 = dtime(TM(2))
DO tile2I = 1, 257, 160
  DO tile2J = 1, 257, 72
    DO tile1K = 1, 257, 16
      DO tile1I = tile2I, MIN((tile2I + 159), 257), 80
        DO J = tile2J, MIN((tile2J + 70), 256), 2
          DO K = tile1K, MIN((tile1K + 12), 254), 4
            mi0 = BB(K, J)
            mi1 = BB(K + 3, J + 1)
            mi2 = BB(K + 3, J)
            mi3 = BB(K, J + 1)
            mi4 = BB(K + 2, J + 1)
            mi5 = BB(K + 2, J)
            mi6 = BB(K + 1, J)
            mi7 = BB(K + 1, J + 1)
          END DO
          DO I
            CC(I, J) = (CC(I, J) +(AA(I, K) * mi0))
            CC(I, J + 1) = (CC(I, J + 1) +(AA(I, K) * mi3))
            CC(I, J) = (CC(I, J) +(AA(I, K + 1) * mi6))
          END DO
        END DO
      END DO
    END DO
  END DO
END DO
```

```
END DO
END DO
END DO
preg = dtime(TM(2))
TIME = preg
FLOP_RATE = (3.3949184E+07 / preg)
WRITE(6, '(-- TEST1: CPU SECONDS = ',F9.6," FLOP_RATE = ",2PE12.5," --")') TIME, FLOP_RATE
PRINT *, CC(21 - 1, 13 - 1), CC(192 - 1, 258 - 1)
STOP
END ! MAIN

k2% f77 -c -R14000 -O3 -S test1.f
k2% grep -i swps test1.s | more
#<swps>
#<swps> Pipelined loop line 14 steady state
#<swps> 40 estimated iterations before pipelining
#<swps> 2 unrollings before pipelining
#<swps> 22 cycles per 2 iterations
#<swps> 32 flops ( 72% of peak) (madds count as 2)
#<swps> 16 flops ( 36% of peak) (madds count as 1)
#<swps> 16 madds ( 72% of peak)
#<swps> 16 mem refs ( 72% of peak)
#<swps> 3 integer ops ( 6% of peak)
#<swps> 35 instructions ( 39% of peak)
#<swps> 2 short trip threshold
#<swps> 7 ireg registers used.
#<swps> 21 fgr registers used.
#<swps>
#<swps>
#<swps> Pipelined loop line 14 steady state
#<swps> 40 estimated iterations before pipelining
#<swps> 2 unrollings before pipelining
#<swps> 13 cycles per 2 iterations
#<swps> 8 flops ( 30% of peak) (madds count as 2)
#<swps> 4 flops ( 15% of peak) (madds count as 1)
#<swps> 4 madds ( 30% of peak)
#<swps> 10 mem refs ( 76% of peak)
#<swps> 3 integer ops ( 11% of peak)
#<swps> 17 instructions ( 32% of peak)
#<swps> 2 short trip threshold
#<swps> 7 ireg registers used.
#<swps> 11 fgr registers used.
#<swps>
#<swps>
#<swps> Pipelined loop line 14 steady state
#<swps> 40 estimated iterations before pipelining
#<swps> 2 unrollings before pipelining
#<swps> 17 cycles per 2 iterations
#<swps> 16 flops ( 47% of peak) (madds count as 2)
#<swps> 8 flops ( 23% of peak) (madds count as 1)
#<swps> 8 madds ( 47% of peak)
#<swps> 16 mem refs ( 94% of peak)
#<swps> 3 integer ops ( 8% of peak)
#<swps> 27 instructions ( 39% of peak)
#<swps> 2 short trip threshold
#<swps> 5 ireg registers used.
#<swps> 17 fgr registers used.
#<swps>