# Code Optimization and Parallelization on the Origins

# - Looking from Users' Perspective

Yan-Tyng Sherry Chang
NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field, CA, USA

**ABSTRACT:** Parallel machines are becoming the main compute engines for high performance computing. Despite their increasing popularity, it is still a challenge for most users to learn the basic techniques to optimize/parallelize their codes on such platforms. In this paper, we present some experiences on learning these techniques for the Origin systems at the NASA Advanced Supercomputing Division. Emphasis of this paper will be on a few essential issues (with examples) that general users should master when they work with the Origins as well as other parallel systems.

## 1. Introduction

Parallel machines such as the IBM SP, the Compaq Alpha servers, the SGI Origins 2000 and 3000, etc., are becoming the main compute engines for high performance computing. For example, for the past few years the NASA Advanced Supercomputing (NAS) Division at NASA Ames has been working closely with SGI to bring increasingly larger single system image Origins into production. At present, many large Origin 2000 (O2K) machines (of sizes 64p, 128p, 256p, and 512p) at NAS are used by scientists for production work. Upgrade of one of these O2K production machines into an O3K is currently underway. In addition, the first 1024-processor single system image shared memory O3K machine, designed under collaborative effort between SGI and NAS, was built in the summer of 2001. This machine has since been subjected to rigorous testing and improvement by NAS and SGI staff in order to deliver the best possible performance for large applications and to bring this machine into production.

Despite the increasing popularity of parallel machines, it is still a challenge for most users to learn how to work with them in order to achieve good performance for their codes. In contrast, obtaining good performance on traditional vector systems such as the Cray C90s seems relatively easy and users usually do not have to change their codes much to achieve this goal. The relatively difficult task of optimizing codes on the parallel machines is rooted in the complexity of the hardware and software (operating system, compilers, libraries, tools, etc.) design driven by the architecture of these systems. In the following, some experiences are presented on learning the optimization and parallelization techniques for the Origins at NAS. Although the discussions are limited to the SGI Origins, many concepts described in this paper should be applicable to other parallel systems as well.

## 2. Users' Perspective

For the O2K and O3K machines, there are quite a few sources of information provided by SGI where users can obtain knowledge about them, including: (1) fee-based training and/or workshops, (2) the annual developer's conference, and (3) SGI on-line library. Among them, the SGI document 007-3430-002, entitled "Origin 2000 and Onyx Performance Tuning and Optimization Guide", is one that is easily accessible to all and provides very thorough information. In addition, most of the information in that document is also applicable to the O3K machines. However, learning from that document can be an enormous task since it is very extensive (with more than 300 pages) and consequently, readers can get lost easily. Furthermore, that document lacks detailed explanation of one of the most fundamental and critical issues, namely, the cache structure, which is a major bottleneck in learning to work with parallel systems.

The important role of "cache" is witnessed in the steps involved in code optimization and parallelization. These steps, as recommended by SGI, are:

For single-processor codes
1. Get the right answers
2. Use existing tuned code (-lm, -lfastm, -lscs)
3. Find out where to tune
4. Let the compiler do the work
5. Tune cache performance

For multi-processor codes
6. Tune for 1-CPU first
7. Parallelize codes
8. Identify and solve bottlenecks
9. Fix false sharing
10. Tune for data placement

For single-processor codes, in step 1, 'get the right answers' on the Origins implies that users have to take care of some porting issues such as data formats, availability of certain libraries, compiler flags to use, and round-off differences, etc. In addition, debugging may be required and it may be a little tedious. Otherwise, step 1 is straight-forward.

In step 2, 'use existing tuned code' refers to using SGI's math (-lm, -lfastm) and scientific libraries (-lscs) in which many routines have been tuned for the Origin's hardware. For example, the standard math library includes special "vector intrinsics" (i.e., vectorized version of certain functions) which

take advantage of software pipelining capabilities of R10000 and R12000 CPUs to fill instruction units in the operation. In addition, many routines in the Scientific Computing Software Library (SCSL) have been optimized for good cache performance. This step does not require user-intervention and thus is straight-forward.

In step 3, 'find out where to tune', usually the profiling tools perfex or SpeedShop are used to analyze the performance of a code and to help identify what a code is suffering and where it occurs. The difficulties in this step are two-fold. First, for a general user, understanding how perfex and SpeedShop work and what their outputs reveal is a non-trivial task. Second, many codes running on the Origins may be identified as not cache-friendly. Yet, a user may not know why it occurs and how to resolve it. Fortunately, some of the bad cache performance problems can be resolved by the compilers automatically.

In step 4, 'let the compiler do the work', the optimization done at -O2 may relieve some of the cache performance problems but it is not as effective as -O3. At -O3, software pipelining (-SWP) and loop nest optimizations (-LNO) are used automatically to improve cache and instruction scheduling. Many techniques in LNO such as padding, loop interchange, loop fusion, cache blocking and prefetching, etc., are very useful for improving cache performance. However, depending on the nature of the code, the settings in -O3 may need to be fine-tuned for ultimate performance. It is also possible that -O3 may not be the right choice for a code. In these cases, users may want to go to step 5 and tune the cache performance themselves.

In step 5, 'tune cache performance' manually is required when the compiler does not or dare not optimize the code for best cache performance. The same techniques used in -O3 can be applied manually. To do this step successfully, it is important that users first have a thorough understanding of the cache structures, the principles and techniques of good cache performance, the nature of his/her own code and the profiling tools to help diagnose cache problems. For general users, this is the major bottleneck in working with the Origins.

For multiple-processor codes, as shown in step 6, the first thing to do is to make sure the code has been optimized for running with a single processor.

In step 7, users then try to parallelize the code as much as possible using either the MIPSpro Auto-Parallelizing Option (APO) through the compiler or various parallel programming models such as OpenMP, MPI or some sort of multi-level parallelism approach.

In step 8, 'identify and solve bottlenecks', users first analyze the performance of the parallel code using simple metrics such as the speedup or the scaling factor (in terms of wall-time used or MFLOPS) to see if the code has been properly parallelized. If the parallel performance is far from ideal, SGI's perfex and SpeedShop or other vendor's profiling tools, (such as CAPTOOL that performs data dependency analysis of a code) can be used to diagnose the problems. For example, is the fraction of the code running in serial negligible compared to the fraction running in parallel? If the serial fraction is significant, try to parallelize it if possible. Another question to ask is whether the loads among the processes have been well-balanced. Keep in mind that the overall performance of a code is determined by the slowest process. If load imbalance is a problem, it may be resolved by redistributing the work load of each process.

In step 9, one examines if false sharing is a problem. False sharing is tied up with the concepts of cache coherency and cache contention among different processors running a parallel program. Understanding these concepts also requires knowledge of the cache structure. False sharing is best diagnosed with the hardware event counters 29 and 31.

In step 10, if all of the above have been tried and the performance is still bad, one examines if poor data placement is the cause. A few tools are described later in this paper that can help diagnose this problem.

In summary, code optimization and parallelization on the Origins, as well as many other parallel systems, is a multi-faceted task which includes working with both the hardware (CPU, counters, cache, memory, etc.) and software (parallel programming models, compilers, debuggers, libraries, tools, etc.). For general users, trying to learn everything at once can be very confusing and frustrating. Instead, a thorough understanding of a few key issues is a better approach, and it provides a better foundation for learning other issues. The few key issues to master, in my own view, include: (1) the memory hierarchy, (2) the cache structure, (3) cache coherency and cache contention, and (4) non-uniform memory access, page and data placement. In this paper, descriptions of these key concepts and a few examples to further clarify them are provided to assist users in overcoming these learning bottlenecks.

## 3. The Memory Hierarchy

The building blocks of the Origins are the "nodes". Each node contains multiple CPUs (up to 2 for O2K; up to 4 for O3K), a secondary cache (L2-cache) external to each CPU and some memory. Nodes are connected by some inter-connect and the communication among them are regulated by the 'hub' (used in O2K) or "Bedrock memory controller" (used in O3K) in each node. Within each CPU, there are also the registers and the primary cache (L1-cache).

During processing, the CPU can only use data in registers, and it can load data into registers only from the primary cache. So data must be brought into the primary cache before it can be used in calculations. The primary cache can obtain data only from the secondary cache, so all data in the primary cache is simultaneously resident in the secondary cache. The data in the secondary cache is obtained from main memory. The memory location of the desired data can be in local memory, remote memory or disk. **Table 1** shows the memory hierarchy, the relative total capacity and the memory access latency (in CPU clock cycles) at each level. The values of the capacity and latency listed are approximate and vary for systems with different hardware. Nevertheless, all Origins exhibit the same trend. That is, with each increasing level, the capacity becomes

larger since the hardware becomes cheaper and more affordable. Yet at the same time, the latency of data access becomes greater (cheaper but slower hardware). A miss at each level of the memory hierarchy multiplies the latency by an order of magnitude or more. Thus, tuning a code to be cache-friendly (meaning diminishing/reducing accesses to main memory, especially remote memory or disk such that all/most accesses are satisfied by caches) is an important factor for getting good performance on the Origins.

| Memory Level | Total Capacity | Maximum Latency (in CPU cycles) |
|---|---|---|
| registers | Bytes | 0 |
| L1 cache | KB | 2-3 |
| L2 cache | ~MB | 8-10 |
| Local memory | ~GB | 75-200 |
| Remote Memory | <TB | > 200 |
| disk | >TB | Long long time |

Table 1. Memory Hierarchy on the Origins

## 4. The Cache Structure

A cache-friendly code exhibits characteristics of minimum accesses to main memory and nearly 100% hit rate on caches. To get the most benefit when data is already in cache, one should exercise two guidelines. The first one is "temporal locality" which means that one should use a cache line intensively while it's in cache and not return to it after it has been written back to memory. The reason for practicing temporal locality is to avoid 'paying' the higher latency more than once. The second guideline is "spatial locality" which means that one should use every word in the same cache line while it is in cache. The reason for practicing spatial locality is due to the nature of a cache line explained below.

### *4.1 Cache Line and Cache Size*

| Cache Type | Cache Size | Cache Line | # of Cache Lines |
|---|---|---|---|
| L1-instruction | 32 KB | 64 B | 512 |
| L1-data | 32 KB | 32 B | 1024 |
| L2-unified | 4 MB (R10000)<br>8 MB (R12000) | 128 B | 32768;<br>65536 |

Table 2. Cache Structure on the Origins

A cache line is the unit of transfer between the main memory and L2 cache or between L2 and L1 caches. On the Origins, each cache line of the L2 cache is 128 bytes long. This is equivalent to 32 words if each word is 4 bytes long. The L2 cache is used jointly for instructions and data. For L1 cache, the instruction cache and the data cache are separate. Each cache line is 64 bytes long for L1-instruction cache and 32 bytes (= 8 four-byte words) for L1-data cache. The consequence of having more than 1 word in each cache line is that when a data is accessed, its nearby consecutive data (ex: the other 7 words in an L1 cache line) are accessed automatically, thus tuning for spatial locality is an important practice.

**Table 2** lists the total size of each cache, the length of a cache line and the corresponding number of cache lines for L1-instruction, L1-data, and L2-unified caches. On the Origins, the L2-cache size is typically 4MB for the R10000 processors and 8MB for the R12000 processors.

### *4.2 Two Way Set Associative*

Before a data can be brought in from memory, a mapping has to be done to determine where the data should reside in cache. Since the size of a cache is much smaller than the size of main memory, many memory locations will map to an identical cache location. There are two extreme mapping methods. The first one is called a 'direct mapping method' in which data at a given memory location can only map to one specific cache location in one specific cache line. The benefit of this method is that it is straight-forward for the processor to keep track of data in cache. If the data needed is not found in its designated cache line, then one can make the conclusion that the data is not in cache. The drawback of this method is that if two or more sets of data that map to the same cache line are needed consecutively, thrashing occurs such that data in the same cache line is always being flushed back to memory, resulting in bad performance due to excessive waste in the form of memory latency.

The second extreme is called a 'fully associative mapping method' in which data can map to ANY cache line in cache. The benefit is that the likelihood of thrashing is reduced to a minimum. The drawback is that it is much more complicated and costly for the processor to check all cache lines for either the desired data or an empty or least-recently used cache line to load in the new data.

The mapping method used on the Origins is two-way set associative'. It is a compromise between the two extremes described above. With two-way set associativity, cache is divided into two 'ways' or parts and the set associativity restricts each data (at a memory location) to a "set" (also called a congruence class) of two cache lines (one in each way) within its own set. The algorithm for determining which set a data should map to is as follows:

Address of data/(cache size of each way)
= cache tag, remainder m
m/ cache line size
= set number, remainder n

For example, assuming the L2 cache size is 4MB, then the cache size of each way is 4MB/2 way = 2MB = 2**21 bytes. The size of an L2 cache line is 128B=2**7 Bytes. The memory on the Origins is byte-addressable. Thus, in binary

representation, the (address of a data)/2**21 gives the value of the cache tag represented by the high bits (bits 21 and above). The value in the lower 21 bits (bits 0-20) is the remainder m. The (value of m)/(2**7) determines which set of the cache (set number), represented by the value in bits 7-20. The value in the lowest bits (bits 0-6) is the remainder n and it determines the byte-location in the cache line this data should reside. Thus, if the addresses of two data have identical value in bits 7-20, these two data will map to the same set of cache lines. Because of the many-to-one mapping between memory and cache set, the cache tag is used by the processor to determine if the data residing in a cache line are those needed by the processor.

### *4.3 Least Recently Used Policy*

Since the set of two cache lines that can be used by a data is determined by the middle bits 7-20, any other data with the same middle bit value also map to the same set. If two data that map to the same set are needed, and, assuming they are not near each other in memory, then both of them can be loaded into cache when the associated cache lines are empty. One of them is loaded into one cache line and the other data will be loaded into the other cache line in the same set. When a third data that also maps to the same set is needed, one of the occupied cache lines has to be flushed in order to load the third data. The rule used on the Origins for determining which of the two cache lines should be flushed is such that the least recently used (read or written) line is selected.

### *4.4 Quiz*

Assume the L2 cache is two-way set associative, its size is 4MB and each cache line is 128B long,

(1) If the memory locations between variable a and variable b are exactly 2MB apart, do they map to the same set?
(2) If another variable c is exactly 2MB apart from b, does c map to the same set as a and/or b?
(3) If a, b, and c are needed one after another, does thrashing occur?
(4) If the L2 cache size is 8MB, while the rest are the same, what are the answers to questions 1,2 and 3?
(5) If everything stays the same as in (4) except that the L2 cache is now four-way set associative, what are the answers to questions 1, 2 and 3?

Assume the L1 cache is two-way set associative, its size is 32KB and each cache line is 32B long,

(6) Do a, b, and c map to the same L1 cache set?
(7) If variable d is 32B away from a, does d map to the same L1 cache set as a?

### *4.5 Practices of Good Cache Use*

Programs that exhibit temporal locality and spatial locality should achieve good performance. In practice, one should follow these guidelines when designing his/her programs:

a. Use stride-1 accesses
b. Avoid power-of-2 sized arrays or do padding
c. Group together data used at the same time

The MIPSpro compilers used on the Origins provide many options to tune cache performance. Among them, the optimization level at -O3 provides software pipelining (-SWP) and loop nest optimization (-LNO) that can better schedule instructions and data for cache performance. The major optimizations by LNO include:

a. array padding
b. loop interchange
c. loop unrolling
d. cache blocking
e. loop fusion
f. loop fission
g. prefetching
h. gather-scatter
i. vector intrinsics

The man page for LNO provides more information about these techniques. Some of these techniques can also be applied manually if the compiler fails to tune a code for cache performance or if the higher level of optimization at -O3 is not suitable for a code.

### *4.6 Detecting Cache Performance Problem*

Two profiling tools, perfex and SpeedShop, provided by SGI can be used to diagnose performance problems. Perfex is generally used to get a quick diagnosis of what the code may be suffering. SpeedShop is used to find out where (which subroutine, function, line number) the bottleneck occurs.

#### *4.6.1 Using perfex*

Using the profiling tool perfex, the values of a few hardware event counters listed below are good indicators of poor cache performance. Among them, the miss handling table occupancy (counter 4) is available only for the R12000 CPUs. Perfex also provides some useful statistics and ranges of estimated time for each event if the -y option is used. Some of the statistics also provide indication of cache performance.

Hardware counters:

a. event counter 26 - secondary data cache misses
b. event counter 25 - primary data cache misses
c. event counter 10 - secondary instruction cache misses
d. event counter 9 - primary instruction cache misses
e. event counter 23 - TLB misses
f. event counter 4 - Miss Handling Table occupancy

Useful perfex statistics for cache performance diagnostics:

a. Primary cache line reuse
This is the number of times, on average, that a primary data cache line is used after it has been moved into the cache. It is calculated as graduated loads plus graduated stores minus primary data cache misses, all divided by primary data cache misses.

b. Secondary Cache Line Reuse
This is the number of times, on average, that a secondary data cache line is used after it has been moved into the cache. It is calculated as primary data cache misses minus secondary data cache misses, all divided by secondary data cache misses.

c. Primary Data Cache Hit Rate
This is the fraction of data accesses that are satisfied from a cache line already resident in the primary data cache. It is calculated as 1.0 - (primary data cache misses divided by the sum of graduated loads and graduated stores).

d. Secondary Data Cache Hit Rate
This is the fraction of data accesses that are satisfied from a cache line already resident in the secondary data cache. It is calculated as 1.0 - (secondary data cache misses divided by primary data cache misses).

e. Cache misses in flight per cycle (average)
This is the count of event 4 (Miss Handling Table (MHT) population) divided by cycles. It can range between 0 and 5 and represents the average number of cache misses of any kind that are outstanding per cycle.

To use perfex, no recompile is needed. When the perfex command is executed, profiling will take place and the output will be sent to stderr. For example,

```
perfex -a -x -y ./a.out
or
perfex -e 26 ./a.out
```

### *4.6.2 Using SpeedShop*

SpeedShop is the generic name for an integrated package of performance tools to run performance experiments on executables, and to examine the results of those experiments. SpeedShop provides several profiling experiment types:

a. User time
b. PC sampling
c. Ideal time (changed to "bbcounts" in SpeedShop 1.4.3)
d. Hardware counter profiling
e. mpi profiling
f. Floating point exception tracing
g. Heap tracing
h. I/O tracing

Brief descriptions of a few commonly used experiments are provided below:

usertime: It returns CPU time, the time your program is actually running plus the time the operating system is performing services for your program. It uses statistical callstack profiling with a time sample interval of 30 milliseconds.

[f]pcsamp: It returns the estimated CPU time consumed by each source code line, machine code line, and function in your program. It uses statistical PC sampling with a sample interval of 10 milliseconds. If the optional f prefix is specified, a sample interval of 1 millisecond is used.

Ideal: It returns the best possible time that a program is capable of achieving. It uses basic-block counting, done by instrumenting the executable.

[f]dsc_hwc: This is one of the many options from hardware counter profiling. It uses statistical PC sampling, based on overflows of the secondary data-cache miss counter (counter 26), at an overflow interval of 131. If the optional f prefix is used, the overflow interval will be 29.

To use SpeedShop, no recompiling is needed. The command ssrun is used to collect SpeedShop performance data and the command prof is used to analyze and display the data collected by ssrun. For example,

```
ssrun -usertime a.out
prof a.out.usertime.m754877
```

### *4.7 Cache Thrashing Example*

In appendix A, a sample program that exhibits cache thrashing behavior is provided. This program is modified from example 6.5 of the SGI document "Origin 2000 and Onyx Performance Tuning and Optimization Guide". The size of each of the four arrays (ex: real*4 a(1024,1024)) is exactly 4MB, and thus accessing the (i,j) element of each array at the same time causes cache thrashing if the L2 cache size is either 4MB or 8MB. To examine the effect of optimization by the compiler on the performance of this code, it is compiled with -O0, -O1, -O2 and -O3 separately. During execution, 1 processor (R10000 CPU, 250MHz with 4MB L2 cache on an O2K machine) is used.

### *4.7.1 Performance*

The first four columns of **Figure 1** show the user CPU time in units of second used by this program at each level of optimization. As seen in this figure, the performance at -O3 is the best and is due mostly to the better cache utilization at this level of optimization.

A similar performance (compared to that with -O3) can be obtained manually without optimization by the compiler. Modifying the code such that the size of each of the four arrays is 1025*1024 instead of 1024*1024 will prevent a(i,j), b(i,j), c(i,j) and d(i,j) from mapping to the same set of cache lines, and no cache thrashing will occur. This is demonstrated in the last column of **Figure 1** for which the manually modified code is compiled with -O0.

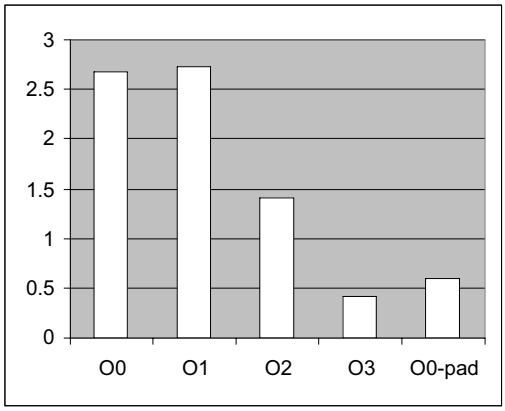


Figure 1. Performance (in CPU seconds) of the program in Appendix A.

### *4.7.2 Diagnosis*

The bad performance of the original code when it is compiled with -O0 is diagnosed with perfex as shown in **Table 3**. With "perfex -a -x -y", it shows that event counter 26 (L2 data cache misses) contributes the most to the time used by the program.

| | event | Counter value | Typical time |
|---|---|---|---|
| 0 | Cycles | 653654512 | 2.614618 |
| 16 | Cycles | 653654512 | 2.614618 |
| 26 | **Secondary data cache misses** | **4178336** | **1.261857** |
| 14 | ALU/FPU progress cycles | 82554064 | 0.330216 |
| 7 | Quadwords written back from scache | 8635504 | 0.221069 |
| 25 | Primary data cache misses | 4348624 | 0.156724 |
| 2 | Issued loads | 29675728 | 0.118703 |
| 18 | Graduated loads | 29524944 | 0.118100 |
| 3 | Issued stores | 29524944 | 0.118100 |
| 19 | Graduated stores | 11248768 | 0.044995 |
| 22 | Quadwords written back from primary data cache | 2475040 | 0.038116 |
| 21 | Graduated floating point instructions | 7787744 | 0.031151 |

Table 3. Some counter values and typical time extracted from perfex -a -x -y output

**Table 4** shows that cache utilization, especially L2, has been improved greatly either by manually padding the arrays or by using a high level of optimization -O3 of the compiler. The location in the original code where the bad cache performance (compiled with -O0) occurs can be detected using ssrun. The results of three experiments are given below to demonstrate how one uses the SpeedShop data to draw some conclusions. These experiments are (1) fpcsamp, (2) ideal, and (3) dsc_hwc (hardware counter 26 for secondary data cache misses). The outputs from prof are quite extensive. For clarity, only data that are relevant to the discussion below are included.

| | **-O0** | **-O3** | **-O0-manual-padding** |
|---|---|---|---|
| L1 Data Cache Misses | 4,348,624 | 20,880 | 361,888 |
| L1 Cache Line Reuse | 8.37 | 1572.17 | 114.06 |
| L1 Cache Hit Rate | 0.89 | 1.0 | 0.99 |
| L2 Data Cache Misses | 4,178,336 | 16 | 81,168 |
| L2 Cache Line Reuse | 0.04 | 129.50 | 3.46 |
| L2 Cache Hit Rate | **0.039** | **0.99** | **0.78** |
| MFLOPS | 2.97 | 24.27 | 15.56 |

Table 4. Comparison of L1 cache, L2 cache and MFLOPS performances

(1) prof -l L2_cache_thrash.fpcsamp.m755026

**Function list, in descending order by time**
secs function (dso: file, line)
2.451 l2_cache_thrashing
(L2_cache_thrash: L2_cache_thrash.f, 1)
0.370 _RANF_4 (libfortran.so: random.c, 154)
**Line list, in descending order by function-time and then line number**
secs function (dso: file, line)
2.339 l2_cache_thrashing
(L2_cache_thrash: L2_cache_thrash.f, 16)

In its "function list" output, it is shown that the program spends 0.370 seconds on calling the random number generator function. It spends 2.451 seconds on the rest of the main program. From the "line list" output, it shows that most time is spent on line 16 (see program in Appendix A) of this program.

(2) prof L2_cache_thrash.ideal.m1838173

**Function list, in descending order by exclusive ideal time**
secs function (dso: file, line)
0.478 L2_cache_thrashing
(L2_cache_thrash: L2_cache_thrash.f, 1)
0.340 _RANF_4 (libfortran.so: random.c, 154)

From this output, it is shown that "ideally", the program, excluding calling the random number generator, should take about 0.478 seconds. This is much shorter than the 2.451

seconds actually used. Thus, it indicates that the program can be improved significantly.

(3) prof -l L2_cache_thrash.dsc_hwc.m1225967

**Function list, in descending order by counts**

```
counts     %      function (dso: file, line)
4159250  100.0 L2_cache_thrashing
                  (L2_cache_thrash: L2_cache_thrash.f, 1)
```

**Line list, in descending order by function-time and then line number**

```
counts     %       function (dso: file, line)
4158988 100.0   L2_cache_thrashing
                  (L2_cache_thrash: L2_cache_thrash.f, 16)
```

This output shows that 100% of L2 cache misses occur in line 16 of the program.

## 5. Cache Coherency and Cache Contention

In a parallel program, a data can be accessed simultaneously by multiple processors. Since the CPU only reads and writes data in its cache, every CPU that accesses the same data has a copy in its own cache. Thus, multiple copies of this data are available from different caches of different processors. If no CPU that has a copy modifies the data (i.e., CPU reads but not writes), each CPU simply uses its own cached copy and no interruption occurs. However, if one CPU modifies a shared data or a shared cache line, this CPU has to become the exclusive owner of the cache line. As a consequence, the copies in the other CPUs become 'invalid' immediately and these CPUs are prevented from using their invalid copies. The new value for this data has to be fetched from the CPU that 'owns' the data after it finishes updating the data. The coordination and synchronization to ensure that all cached copies of data are true reflections of the data in main memory is called cache coherency. On the Origins, maintaining cache coherency is the responsibility of the 'hub' located outside of the CPU.

Cache contention is a phenomenon that multiple CPUs alternatively and repeatedly update the same cache line. Thus each CPU has to become the exclusive owner of that cache line in turn. This will slow down performance dramatically. Data that are mostly read and rarely written do not cause cache contention for parallel programs.

There are two variations of cache contention. The first one is called memory contention in which two or more CPUs try to update the same variables. The second one is called false sharing in which the CPUs repeatedly update different variables that occupy the same cache line. Memory contention usually occurs due to the algorithm of a program. Fixing it may require changing the algorithm. In contrast, false sharing usually occurs unintentionally and can be fixed quite easily.

### *5.1 Detecting Cache Contention*

a. performance does not scale
b. event counter 31 - store or prefetch-exclusive to shared block
c. event counter 29 - external invalidation hits in Scache

When the scaling of a code is poor, one first checks if it is caused by cache contention. The R10000 and R12000 event counter 31 is the best indicator of cache contention between CPUs. The CPU that accumulates a high count of event 31 is repeatedly modifying shared data. Other CPUs that have a copy of the modified cache line will be sent invalidations. Another good indicator of cache contention is event 29. The CPU that produces a high count of event 29 is being slowed because it is using shared data that is being updated by a different CPU. The CPU doing the updating generates event 31.

To get estimates of the counts for event 29 and 31, use
perfex -a -mp ./a.out (or perfex -a -x -y -mp ./a.out)

One can get more accurate counts of these two events by using, for example,

```
perfex -e 29 -mp ./a.out
or
setenv SPEEDSHOP_HWC_NUMBER 29
setenv SPEEDSHOP_HWC_COUNTER_OVERFLOW 99
ssrun -exp prof_hwc ./a.out
prof a.out.prof_hwc.m1234 > prof.m1234.output
```

### *5.2 False Sharing Example*

In appendix B, a sample program that exhibits false sharing behavior is provided. This program is modified from example 8.5 of the SGI document "Origin 2000 and Onyx Performance Tuning and Optimization Guide". When this program is executed with 4 processors, in the subroutine sum85 of this program, s(1) is updated by processor 1 while at the same time, s(2) is being updated by processor 2, and so on. Since the size of s is only 4 words, all elements of s are likely to reside in the same cache line (both in L1 and L2 caches). Thus, each processor has to gain exclusive ownership of this cache line when updating s(i), resulting in false sharing.

#### *5.2.1 Performance*

**Figure 2** shows the performance of this code running with 1, 2 and 4 processors (R10000 CPU, 250MHz with 4MB L2 cache on an O2K machine). The performance is measured as the 'wall-time' (in seconds, reported by t3 in the program) spent on executing the subroutine where s(i) is updated. As seen in this figure, when the code is compiled with -O0 -mp, the code runs slower with 2 or 4 CPUs compared to 1 CPU! The same behavior is seen when the code is compiled with -O1 -mp.

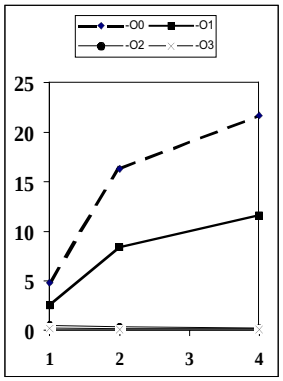


Figure 2(a)

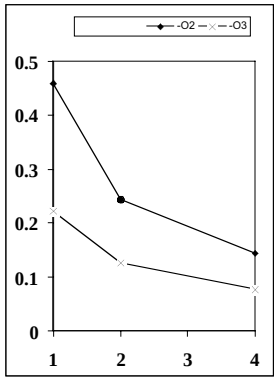


Figure 2(b)

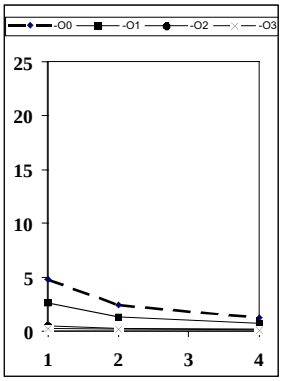


Figure 3(a)

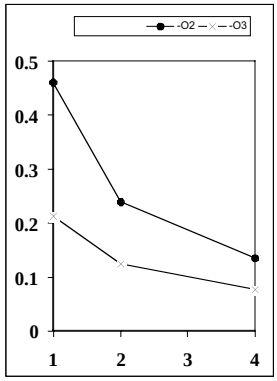


Figure 3(b)

Fortunately, false sharing is fixed for this program by the compiler when -O2 or -O3 is used. As shown in **figure 2(a)** and **figure 2(b)**, with -O2 or -O3, the performance with 2 CPUs or 4 CPUs is better than with 1 CPU. In addition, the performance now scales with the increased number of CPUs used.

Fixing false sharing for this program manually is also quite straight-forward. One can either rewrite the code such that a temporary variable, private to each thread, is used in the place of s(i) or change array s(4) to s(32,4) and replace s(i) with s(1,i) in the subroutine. This second approach provides a padding of 32 words between s(1,1) and s(1,2), where s(1,1) is to be updated by processor 1 and s(1,2) is to be updated by processor 2. Since s(1,1) and s(1,2) are now 32 words apart, they will not reside in the same cache line for either L1 or L2 cache, thus preventing false sharing. Using this second approach, **figure 3(a-b)** shows the performance of the modified code at -O0, -O1, -O2 and -O3 when 1, 2 or 4 CPUs are used. The performance at -O2 and -O3 is almost identical to the original code. But the performance at -O0 and at -O1 with 2 or 4 CPUs has not only improved significantly but also shows proper speedup with the increased number of CPUs used.

### *5.2.2 Diagnosis*

For the original program, the fact that performances do not scale gives an indication that false sharing is occurring (with -O0 and -O1). This can be further validated by using event counters 29 and 31. As a demonstration, 'perfex -e 29 -mp' and 'perfex -e 31 -mp' are used to obtain the counts for each thread.

**Table 5** shows the comparisons of these counter values of the original program (with false sharing) and the modified program (no false sharing). The results shown here are obtained when the two programs are compiled with -O0.

| | **False Sharing** | | **No False Sharing** | |
|---|---|---|---|---|
| | **1 CPU** | **4 CPUs** | **1 CPU** | **4 CPUs** |
| Counter 29 | 3,766 | 482,491 | 4,643 | 1,638 |
| | | 487,530 | | 1,796 |
| | | 539,327 | | 2,060 |
| | | 516,084 | | 2,338 |
| Counter 31 | 0 | 991,489 | 0 | 112 |
| | | 1,020,035 | | 110 |
| | | 479,600 | | 106 |
| | | 535,563 | | 721 |

Table 5. Counts of event 29 and event 31 for the false sharing example

For event counter 29, the values obtained for each thread of the 4 CPU run of the original program (with false sharing) are much larger than that obtained with 1 CPU. They are also much larger than those obtained at both 1 CPU and 4 CPU runs of the modified program with no false sharing. For event counter 31, the values of the two 1 CPU runs (one for the original and one for the modified program) are both zero. The values for each thread of the 4 CPU run of the original program (with false sharing) are much larger than those obtained for each thread of the 4 CPU run of the modified program (no false sharing). Thus, the much larger counts of events 29 and 31 when multiple processors are used for the original program provide definitive proof that false sharing is indeed occurring.

## 6. Non-Uniform Memory Access (NUMA)

For a cache-friendly parallel program, memory requests are satisfied primarily by cache. Thus, the performance is not determined by the actual location of data in physical memory. For non-cache-friendly programs, on the contrary, memory requests are satisfied by main memory, and the placement of data in memory plays a crucial role in performance.

On the Origins, the memory access model used is shared memory NUMA model. In this model the memory is physically distributed amongst the processors, but is globally addressable. This means that a processor can access data that reside in the memory of a different node. The memory is shared, but access time will differ depending on whether the requested memory address is local or remote to the requesting processor. Remote memory access requires communication through the inter-

connect network and thus takes additional time. For the O2Ks, as shown in **Table 6**, the latency to load a cache line from local memory is about 485ns. If the data has to be accessed from a remote memory that is n-hops away (through n-routers), the latency is about (485+n*100) ns. The values listed in **Table 6** do vary depending on the topology and hardware of the memory and inter-connect systems. Nevertheless, it shows that the farther the data, the longer it takes to access it. Tuning non-cache-friendly parallel programs on the Origins often means optimizing the data placement in physical memory to achieve 'memory locality' such that the memory access time is minimized for each process.

| NCPUS | Maximum Route Hops | Latency |
|---|---|---|
| 1 or 2 | 0 | 485 ns |
| 3 or 4 | 1 | 585 ns |
| 5-8 | 2 | 685 ns |
| 9-16 | 3 | 785 ns |
| 17-32 | 4 | 885 ns |
| 33-64 | 5 | 985 ns |

Table 6. Max hops and latency

### *6.1 Page*

For virtual memory systems such as the Origins, memory is allocated by the operating system to a user's program in units of a page. The default page size on the Origins is 16KB. However, a system can be configured to set aside a certain number of larger pages (ex: 64KB, 256KB, 1MB, 4MB, etc. defined in /var/sysgen/stune) to be used by programs that request them. The command 'osview' (select option 3) allows you to see how many pages of each large page size (>16KB) are available (not being used yet) in each node.

A frequently overlooked concept regarding pages is that a page is the "smallest continuous memory" that the operating system can allocate to your program. For example, if the size of an array is exactly n pages, the array elements may be spread among n pages or n+1 pages depending on whether the allocation of this array starts at the beginning of a page. These n or n+1 pages can not be split and allocated among the memory of more than n or n+1 nodes since only a 'whole' page can be allocated in memory. However, the array can still be accessed by as many processors as desired (ex: >> 2n processors on O2K or >> 4n on O3K). In this circumstance, the performance will not scale beyond 2n processors on O2K or 4n processors on O3K if the program is memory intensive.

### *6.2 Data Placement*

For parallel programs, memory locality management (i.e., placing data in or near the local memory of the CPU that needs these data) is an important factor to achieve good parallelism (speedup and scalability). The IRIX operating system is capable of taking care of many memory locality needs automatically.

When necessary, users can fine-tune data placement by (i) choosing a specific data placement policy through MP library environment variables, (ii) enabling page migration, (iii) adding certain compiler directives in the program, (iv) using the dplace tool, and (v) adding runtime calls to specific library routines. In the following, the two most common approaches, namely, using the first touch placement policy and the round-robin placement policy are described.

The default data placement policy on the Origins is 'first touch placement'. With this policy, memory for a page is allocated in the node whose CPU(s) first touches this page. If this is not possible, this page is allocated in a node as close as possible to the CPU that first touches the page. This placement policy is usually good for programs that exhibit good locality to a processor.

A common practice of initializing arrays by a single CPU with the first touch placement policy often creates bottlenecks for the subsequent computational crunch. This is due to the fact that (1) memory references by most other CPUs will be non-local and (2) traffic to the memory of the single node containing the arrays is jammed by the requests from many CPUs. A simple remedy is to parallelize the initialization step so that data can be distributed to the memory of many CPUs that first touch each page.

With round-robin placement, pages are distributed to memory in many nodes in a round-robin fashion. The benefits of this placement policy are two-fold. First, data will be distributed 'randomly' (though not optimal) and thus will be good for programs where all CPUs tend to access all data equally. Second, memory access will not be targeted to a single node thus avoiding a bottleneck.

To use round-robin placement policy, no modification to the code is required. One simply sets the following environment variable before executing the program:

```
setenv _DSM_ROUND_ROBIN
```

### *6.3 Quiz*

Assume on an O2K, there are two CPUs in each node and on an O3K, there are four CPUs in each node. The default page size is 16KB per page.

(1) Using an O2K machine, if the size of array a is 48KB, in the following program, how would array a be allocated in memory if one sets
setenv OMP_NUM_THREADS 3?

```
real a(48KB)
!$OMP PARALLEL DO
do I=1,n
initialize a(i)
end do
```

(2) If everything is same as (1) except one sets
setenv OMP_NUM_THREADS 4, how would array a be allocated in memory?

(3) If everything is same as (1) except that an O3K machine is used, how would array a be allocated in memory?

(4) If everything is same as (1) except that the page size has been changed to 64KB per page, how would array a be allocated in memory ?
(5) If everything is same as (1) except that the OpenMP directive in the program is removed and one then sets setenv _DSM_ROUND_ROBIN, how would array a be allocated in memory ?

### *6.4 Detecting Memory Placement Problem*

a. Poor scaling
b. dlook
c. dsm_home_threadnum
d. ssrun -numa

One would suspect memory placement problems if everything else (ex: cache contention, load imbalance, etc.) has been checked and fixed and yet scaling is still poor. A few tools are available for examining memory placement. The tool dlook provides an easy way to examine how data (stack and heap data with page size information) are placed when processes exit. If sampling is enabled, (by using - sample n), data is also displayed every n seconds. To obtain a dlook output, use the -out option:

```
dlook -out dlook_output ./a.out
```

In addition to dlook, one can obtain more detailed information about data placement by using the dsm_home_threadnum() intrinsic within the program. This function takes an address as an argument, and returns the number of the CPU in whose local memory the page containing that address is stored. It is used in Fortran as follows:

```
integer dsm_home_threadnum
numthread = dsm_home_threadnum(array(i))
```

Since two CPUs are connected to each node of an O2K and they share the same memory, dsm_home_threadnum returns the lower CPU number of the two running on the node with the data. Note that the numbering of CPUs reported by dsm_home_threadnum is relative to the program, not the number of absolute physical CPU. The benefit of using dsm_home_threadnum() over dlook is that one can find out where each specific data is stored, and with additional engineering of the code, which process is accessing this data. The drawback is that one has to instrument the code manually and it is non-trivial to analyze the information obtained.

In the past, no profiling tool would definitively inform you that poor performance of a code is caused by poor data placement. In the most recent release of SpeedShop (version 1.4.3), a new experiment type (ssrun -numa) is provided that allows much easier diagnosis of memory placement problems. The output of this experiment reports the percentage of remote memory access (number of remote memory accesses/total memory access sampled) and the average cc-NUMA routing distance. To use this experiment, do, for example:

```
ssrun -numa ./a.out
prof a.out.numa.m1234 > a.out.numa.m1234.out
```

### *6.5 Memory Placement Example*

In appendix C, a sample program is provided for examining the performance of three different memory placement approaches. This program is modified from example 8-7 of the SGI document "Origin 2000 and Onyx Performance Tuning and Optimization Guide".

The three different approaches are:

(a) Use the program in appendix C as is. The initialization loop is not parallelized.
(b) Use the program in appendix C as is. The initialization loop is not parallelized. However, "setenv _DSM_ROUND_ROBIN" is used during run-time.
(c) Modify the program in appendix C such that the initialization loop is parallelized with OpenMP directive "!$OMP PARALLEL DO private(i) shared(a,b,c,d)". _DSM_ROUND_ROBIN is not set in this case.

The original and modified codes are compiled with -O3 -mp. For each of the three approaches, the variable "niters" is read in from stdin as 100. During run time, different numbers of CPUs are used by setting "setenv OMP_NUM_THREADS n" where n varies from 1 to 32. An O2K machine with 250MHz R10000 CPU, 4MB L2 cache, and 490MB memory per node is used during execution.

**Figure 4** shows the performances of each of the three approaches. In **figure 4(a)**, the real time (y-axis), in seconds as obtained from the function dtime and reported as t2 in the program, spent on the initialization is plotted as a function of n (x-axis) set by OMP_NUM_THREADS (which is same as NCPUS for the whole program). As seen in this figure, for the first two approaches using the original code, the real time is roughly the same no matter what the value of n is. This is expected since only 1 processor is responsible for initializing the arrays with either first-touch or the round-robin placement policy. For the third approach where the initialization loop is parallelized, the time spent on initialization decreases gradually as more processors are added.

In **figure 4(b)**, for each of the three approaches, the real time (y-axis), reported as t3 in the program, spent on real work is plotted as a function of n (x-axis) set by OMP_NUM_THREADS. For the first approach, where data are most likely localized to the node whose CPU first touches all pages, the performance is better using 2 CPUs than 1 CPU. This is because the memory references are still local for the two CPUs in a single node. However, increasing the number of CPUs beyond 2 does not decrease the wall-time any more since memory references will be non-local for any CPUs other than the first two.

For the second approach, data are placed in the memory of many nodes in a round-robin fashion. For this program, as seen in **figure 4(b)**, the wall-time spent on real work decreases as

more CPUs (and thus, more memory in different nodes are available for data placement) are added. Similar behavior is seen for the third approach. In addition, the performances using the third approach are better than the second due to more optimum data placement.

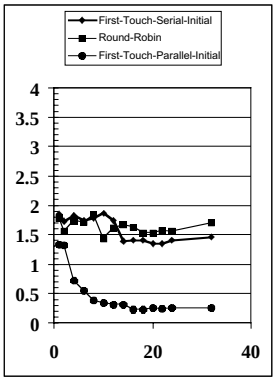


Figure 4(a)

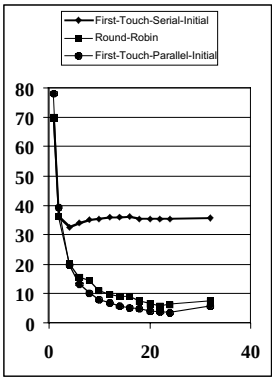


Figure 4(b)

## 7. Conclusion

Code optimization and parallelization on the Origins as well as other parallel systems is a multi-faceted task. Understanding the few basic issues described in this paper is essential for building a solid foundation and for easier transition to learning other related subjects. Training or workshops provided by vendors or others are good sources for getting an overview of many relevant topics. However, attending these sessions is usually not adequate. For the Origins, the SGI document "Origin 2000 and Onyx Performance Tuning and Optimization Guide" is a good reference to keep handy. Many examples provided in this document are quite useful. In addition, it is important to (1) do more hands-on experiments, and (2) when things do not work "as expected", find out the cause(s) of the "misbehavior". It is through these hands-on experiments and thorough investigation that the author finds most effective in learning to proceed with code optimization and parallelization on parallel systems.

## 8. Acknowledgments

Support from the NASA Advanced Supercomputing Division under contract numbers ARC320.000.2 and ARC330.000.2 is acknowledged.

## 9. About the Author

Sherry Chang received her Ph.D. in Theoretical Chemistry from the University of California at Berkeley in 1991. Before joining NASA, she was a research scientist at the Molecular Research Institute performing research in areas of quantum chemistry, molecular dynamics and homology modeling of biological systems. At present, she is a Scientific Consultant at the NASA Advanced Supercomputing Division of NASA Ames Research Center. E-mail: schang@nas.nasa.gov

## 10. Appendix

### *A. Cache Thrashing Sample Program*

```
1 program L2_cache_thrashing
2
3! Arrays a, b, c, and d are all 4MB in size. Accessing a(i,j),
b(i,j)
4! c(i,j), and d(i,j) simultaneously causes L2_cache_thrashing
when
5! the size of L2 cache is either 4MB or 8MB
6! with 2-way set associativity
7
8 dimension a(1024,1024), b(1024,1024), c(1024,1024),
d(1024,1024)
9
10 call random_number(b)
11 call random_number(c)
12 call random_number(d)
13
14 do j=1,1024
15 do i=1,1024
16 a(i,j)=b(i,j)+c(i,j)*d(i,j)
17 end do
18 end do
19
20 write (12) a
21
22 stop
23 end
```

### *B. False Sharing Sample Program*

```
program false_sharing
parameter (m=4,n=100000)
real a(n,m),s(m)
real*4 dtime,tarray(2)
t1=dtime(tarray)
do i=1,m
do j=1,n
a(j,i)=(i+j)/5000.0
end do
end do
t2=dtime(tarray)
do k=1,100
call sum85(a,s,m,n)
write (6,*) 'k= ',k
write (6,*) s
end do
t3=dtime(tarray)
print *, 'time on initialization = ', t2
print *, 'time on real work = ', t3
stop
end
subroutine sum85 (a,s,m,n)
integer m, n, i, j
```

```
real a(n,m), s(m)
!$omp parallel do private(i,j), shared(s,a)
do i = 1, m
s(i) = 0.0
do j = 1, n
s(i) = s(i) + a(j,i)
enddo
enddo
return
end
```

### *C. Memory Locality Sample Program*

```
program memory_locality
      integer i, j, n, niters
      parameter (n = 8*1024*1024,ndim = n+35)
      real a(ndim),b(ndim),c(ndim),d(ndim),q
      real*4 dtime,tarray(2)

      read *, niters
      print *, ' niters = ', niters

! initialization
     t1=dtime(tarray)
! comment 1
do i = 1, n
        a(i) = 1.0 - 0.5*i
        b(i) = -10.0 + 0.01*(i*i)
        c(i) = 2*i - 0.3
        d(i) = 0.5*i
      enddo
     t2=dtime(tarray)

! real work
      do it = 1, niters
        q = 0.01*it
!$omp parallel do private(i) shared(a,b,c,d,q)
        do i = 1, n
          a(i) = a(i) + q*b(i)
          c(i) = c(i) + d(i)
          a(i) = a(i) + sqrt(c(i))
        enddo
       call sub(a,b,ndim)
      enddo

      t3=dtime(tarray)
      print *, a(1), a(n), q
      print *, 'time on initialization = ', t2
      print *, 'time on real work =',t3
      end

      subroutine sub(a,b,ndim)
      real a(ndim), b(ndim)
      return
      end
```