Cray SV1™ Application Optimization Guide

S–2312–36

© 2002 Cray Inc. All Rights Reserved. The contents of this document may not be copied or duplicated in any manner, in whole or in part, without the prior written permission of Cray Inc.

U.S. GOVERNMENT RESTRICTED RIGHTS NOTICE

The Computer Software is delivered as "Commercial Computer Software" as defined in DFARS 48 CFR 252.227-7014.

All Computer Software and Computer Software Documentation acquired by or for the U.S. Government is provided with Restricted Rights. Use, duplication or disclosure by the U.S. Government is subject to the restrictions described in FAR 48 CFR 52.227-14 or DFARS 48 CFR 252.227-7014, as applicable.

Technical Data acquired by or for the U.S. Government, if any, is provided with Limited Rights. Use, duplication or disclosure by the U.S. Government is subject to the restrictions described in FAR 48 CFR 52.227-14 or DFARS 48 CFR 252.227-7013, as applicable.

Autotasking, CF77, Cray, Cray Ada, Cray Channels, Cray Chips, CraySoft, Cray Y-MP, Cray-1, CRInform, CRI/*Turbo*Kiva, HSX, LibSci, MPP Apprentice, SSD, SuperCluster, UNICOS, UNICOS/mk, and X-MP EA are federally registered trademarks and Because no workstation is an island, CCI, CCMT, CF90, CFT, CFT2, CFT77, ConCurrent Maintenance Tools, COS, Cray Animation Theater, Cray APP, Cray C90, Cray C90D, Cray CF90, Cray C++ Compiling System, CrayDoc, Cray EL, Cray Fortran Compiler, Cray J90, Cray J90se, Cray J916, Cray J932, CrayLink, Cray MTA, Cray MTA-2, Cray MTX, Cray NQS, Cray/REELlibrarian, Cray S-MP, Cray SSD-T90, Cray SV1, Cray SV1ex, Cray SV2, Cray SX-5, Cray SX-6, Cray T90, Cray T94, Cray T916, Cray T932, Cray T3D, Cray T3D MC, Cray T3D MCA, Cray T3D SC, Cray T3E, CrayTutor, Cray X-MP, Cray XMS, Cray-2, CSIM, CVT, Delivering the power . . ., DGauss, Docview, EMDS, GigaRing, HEXAR, IOS, ND Series Network Disk Array, Network Queuing Environment, Network Queuing Tools, OLNET, RQS, SEGLDR, SMARTE, SUPERLINK, System Maintenance and Remote Testing Environment, Trusted UNICOS, and UNICOS MAX are trademarks of Cray Inc.

IBM is a trademark of International Business Machines Corporation. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. X/Open is a trademark of X/Open Company Ltd. The X device is a trademark of the Open Group.

The UNICOS operating system is derived from UNIX System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

New Features

Cray SV1™ Application Optimization Guide S–2312–36

This version adds information on optimizing the Cray SV1ex system.

Chapter 4 has been expanded to include more information about and examples of the optimizations performed by the Fortran and C/C++ compilers on the Cray SV1 series systems.

Record of Revision

Version	*Description*
3.5	January 2001 Original printing.
36	June 2002 This version includes information that supports the Programming Environment 3.6 release.

Contents

Page

Preface **ix**

Related Publications . . . ix

Ordering Documentation . . . ix

Conventions . . . x

Reader Comments . . . xi

Introduction [1] **1**

About This Manual . . . 1

Optimization Overview . . . 2

Hardware Overview . . . 5

The Processor . . . 8

Cache . . . 9

I/O . . . 9

Evaluating Code [2] **11**

CPU-Bound Programs . . . 11

Using the `hpm` Command to Issue Reports . . . 13

Procedure 1: Using the `hpm` Command to Determine Processor Performance Statistics . 13

Analyzing Multistreaming Code . . . 16

The `ps`(1) Command . . . 17

The `jstat`(1) Command . . . 18

The `cpu`(8) Command . . . 18

Memory and Cache . . . 19

Cache . . . 19

Code Size . . . 20

Procedure 2: Determining if the code is memory bound . . . 20

Determining How Much Memory Is Available on Your System . . . 22

Page

I/O Bound . . . 24
Procedure 3: Determining if the Code is I/O Bound . . . 24

Multistreaming [3] **25**

Compiler Options and Directives . . . 26
Fortran Compiler Options . . . 26
C and C++ Compiler Options . . . 27
Directives . . . 27
`concurrent` Directive . . . 27
`preferstream, stream`, and `nostream` Directives . . . 28
Loops That Are Multistreamed by the Compiler . . . 29
Bit Matrix Multiply (BMM) and Multistreaming . . . 30
Tasking and Multistreaming . . . 31
Multitasking on MSPs with Multistreaming . . . 32
Multitasking on MSPs without Multistreaming . . . 32
Vectorization and Multistreaming . . . 33
Analyzing the Performance of a Multistreaming Program . . . 33
The `prof` and `profview` Commands . . . 33
The `MSP_STATS` Environment Variable . . . 35

Optimizing Using Vectorization [4] **39**

What Is Vectorization? . . . 39
Loopmark Listings . . . 41
Vectorization . . . 43
Fully Vectorized Loops . . . 43
Partially Vectorized Loops . . . 43
Conditionally Vectorized Loop . . . 44
Reduction Loop . . . 44
Shortloop . . . 45
Vector Update Loop . . . 45
Computed-Safe Vector Length Loop . . . 46

Page

Vectorization Inhibitors 47
Vectorization and Dependencies 47
Other Loop Optimizations 48
Outer-loop Vectorization 49
Loop Unrolling 49
Loop Interchange 51
Loop Collapse 52
Loop Fusion 54
Pattern Matching 55
Autotasking 56
Streaming 56
Loop Blocking 57

Optimizing Memory Use [5] **59**

Overview of Memory 59
Central Memory 59
Cache 60
Optimizing Cache Use 62
Using Vector Cache Effectively 62
Minimizing Stores 63
Porting Issues 66
Managing Memory 67
Understanding Memory Management 68
Dynamic Heap 69
Dynamic Common Blocks 70
Identifying Large Amounts of Memory Wait Time or System CPU Time 71
Procedure 4: Creating a Report 71
Evaluating Dynamic Memory Alternatives and Applying a Technique 72
Large Number of System Calls 72
Memory Expanded or Contracted in Small Increments 72
Other Reasons for Excessive Memory Activity 73

Page

Temporary Memory Expansion of Significant Duration 74
Heap Blocks Release Order 74
Memory Initialization 74
Loader Directives 74
Optimal Heap Size 75
Procedure 5: Determining Optimal Heap Size 75

Optimizing I/O [6] 79

Optimizing Formatted I/O 79
Changing to Unformatted I/O 79
Reducing the Amount of Formatted I/O 80
Increasing Formatted I/O Efficiency for Fortran Programs 80
Minimizing the Number of Data Items in the I/O List 80
Using a Single `READ`, `WRITE`, or `PRINT` Statement 81
Using Longer Records 81
Using Repeated Edit Descriptors 82
Using Data Edit Descriptors That Are the Same Width as the Character Data 82
Increasing Formatted I/O Efficiency for C++ Programs 82
Increasing Library Buffer Sizes for Formatted I/O Requests 83
Optimizing Large, Sequential, Unformatted I/O Requests 83
Changing I/O File Format to Unbuffered and Unblocked 83
Converting to Asynchronous I/O 84
Using the `assign` Command to Convert Code to Asynchronous I/O 85
Optimizing Asynchronous I/O 85
Example 1: C++ Example of Converting to Asynchronous I/O 86
Using Effective Library Buffer Sizes 86
Optimizing Small, Sequential, Unformatted I/O Requests 87
Using Effective Library Buffer Sizes 87
Increasing I/O Request Size and Issuing Fewer Requests 87
Using the Memory-Resident (MR) FFIO Layer 87
Optimizing Techniques for Direct Access I/O 87

Page

Fortran Direct Access I/O . . . 88
Example 2: Fortran Direct Access . . . 88
C++ Direct Access I/O . . . 88
Example 3: C++ Direct Access . . . 88
Optimizing Techniques for Direct Access Code . . . 89
Optimizing Asynchronous I/O Requests . . . 89
Using Unblocked File Format . . . 90
Avoiding Cache . . . 90
Using Effective Library Buffer Sizes . . . 90
Balancing Workload . . . 91
Minimizing Required Synchronization . . . 91
Tune FFIO User Cache . . . 91
Using an Optimal Storage Device . . . 91
Memory-Resident (MR) Files . . . 92
Memory-Resident Predefined File Systems . . . 92
Disk Striping . . . 92
Disk Arrays . . . 93
Disks . . . 93
Tapes . . . 94
Minimizing System Calls . . . 94
Using the SSD-I on Cray SV1ex Model Systems . . . 94

Glossary **97**

Index **105**

Figures

Figure 1. Optimization Overview . . . 4
Figure 2. Block Diagram . . . 7
Figure 3. GigaRing I/O . . . 10
Figure 4. Evaluating Code . . . 12
Figure 5. Dividing Loop Iterations among CPUs . . . 25

Page

Figure 6. Profview Pie Chart 35
Figure 7. Scalar versus vector, illustrated 40
Figure 8. Optimizing memory-bound code 68
Figure 9. Load map statistics 77

Tables

Table 1. Single-Processor Hardware Expectations 15
Table 2. Determining if the code is dominated by scalar or vector operations 15
Table 3. Loopmark Listing Keys 42
Table 4. Typical Latencies 59

Preface

This document describes techniques you can use to optimize Fortran, C++, or C code on the Cray SV1 series of systems. This manual discusses using the Cray Fortran Compiler, the Cray Standard C Compiler, the Cray C++ Compiler, and various performance tools to help you analyze and optimize your code.

This is a guide for programmers with working knowledge of the Cray Fortran Compiler, Cray Standard C Compiler, or Cray C++ Compiler.

Related Publications

The following documents contain additional information that may be helpful:

- *Cray Fortran Compiler Commands and Directives Reference Manual*
- *Cray Standard C/C++ Reference Manual*
- *Application Programmer's Library Reference Manual*
- *Intrinsic Procedures Reference Manual*
- *Scientific Libraries Reference Manual*
- *Introducing the Cray TotalView Debugger*

Ordering Documentation

To order software documentation, contact the Cray Software Distribution Center in any of the following ways:

E-mail:
`orderdsk@cray.com`

Web:
`http://www.cray.com/craydoc/`

Click on the `Cray Publications Order Form` link.

Telephone (inside U.S., Canada):
1–800–284–2729 (BUG CRAY), then 605–9100

Telephone (outside U.S., Canada):
Contact your Cray representative, or call +1–651–605–9100

Fax:
+1–651–605–9001

Mail:
Software Distribution Center
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items, such as file names, pathnames, man page names, command names, and programming language elements.
variable	Italic typeface indicates an element that you will replace with a specific value. For instance, you may replace *filename* with the name `datafile` in your program. It also denotes a word or concept being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a syntax representation for a command, library routine, system call, and so on.
`. . .`	Ellipses indicate that a preceding element can be repeated.

Reader Comments

Contact us with any comments that will help us to improve the accuracy and usability of this document. Be sure to include the title and number of the document with your comments. We value your comments and will respond to them promptly. Contact us in any of the following ways:

E-mail:
swpubs@cray.com

Telephone (inside U.S., Canada):
1–800–950–2729(Cray Customer Support Center)

Telephone (outside U.S., Canada):
Contact your Cray representative, or call +1–715–726–4993(Cray Customer Support Center)

Mail:
Software Publications
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA

Introduction [1]

This document provides information about techniques you can use to optimize Fortran, C, or C++ code on the Cray SV1 series of systems. This is a guide for programmers with working knowledge of either the UNICOS or UNIX operating system and experience in working with the Cray Fortran Compiler (formerly called the CF90 Compiler), Cray C++ Compiler, or Cray Standard C Compiler.

This document makes the following assumptions about code to be optimized:

- The code has been debugged and is running on a Cray SV1 series system. See the *Introducing the Cray TotalView Debugger* manual for more information on debugging.
- When running the code, you have used the fastest compiler options and data types for your program. For example, you have used aggressive optimization options (`-O scalar3,vector3` for the Fortran `ftn` command or `-hscalar3,vector3` for the C or C++ `cc` or `CC` command) and, if you do not need 96 bits of floating-point precision, you have used single precision (`ftn -dp` or the `float` or `double` type specifier instead of `long double` type specifier within C++ code).

  Note: Unless otherwise noted, all discussions of C++ code optimizations also apply to the C language.

1.1 About This Manual

The following information is provided in this manual:

- Section 1.2, page 2 gives an overview of the optimization process.
- Section 1.3, page 5 provides an overview of the Cray SV1 series system hardware.
- Chapter 2, page 11 describes how to evaluate code and determine where to focus optimization efforts.
- Chapter 3, page 25 describes multistreaming for Cray SV1 series systems. Because this is a major optimization for processor performance, it is in its own chapter.
- Chapter 4, page 39 describes vectorization. Because this is a major optimization for processor performance, it is in its own chapter.

- Chapter 5, page 59 describes how to optimize memory use.
- Chapter 6, page 79 describes how to optimize input and output.

1.2 Optimization Overview

Optimization is the process of changing a program or the environment in which it runs to improve its performance. Performance gains generally fall into one of two categories of measured time:

- *User CPU time*. Time accumulated by a user process when it is attached to a CPU and executing. When running on a single CPU, CPU time is a fraction of elapsed time. When the program is running in parallel, CPU time is a multiple of elapsed time.
- *Elapsed (wall-clock) time*. The amount of time that passes between the start and termination of a user process. Elapsed time includes the following:
  - User CPU time
  - UNICOS system CPU time
  - I/O wait time
  - Sleep or idle time

Figure 1, page 4, shows a comprehensive view of the single-processor optimization process. This is an iterative method that requires you to determine when to stop optimizing the code.

Note: During the optimization process you will need to execute your code repeatedly to assess performance and measure performance gain. To save execution time, we encourage you to work with smaller sample data sets that exercise all of the code within your program.

Each area of optimization involves a three-step method, which includes the following general steps:

1. Identify a problem.
2. Evaluate the problem.
3. Apply a technique.

Check your answers after each code modification to avoid regression, then compare the new performance against the initial performance to decide where to

proceed. If you want to continue optimizing for single-processor performance, return to the initial analysis.

Note: If none of these techniques brings you closer to the desired computation rates for the processor for your system, you may want to seek help from Cray. See your Cray representative for more information.

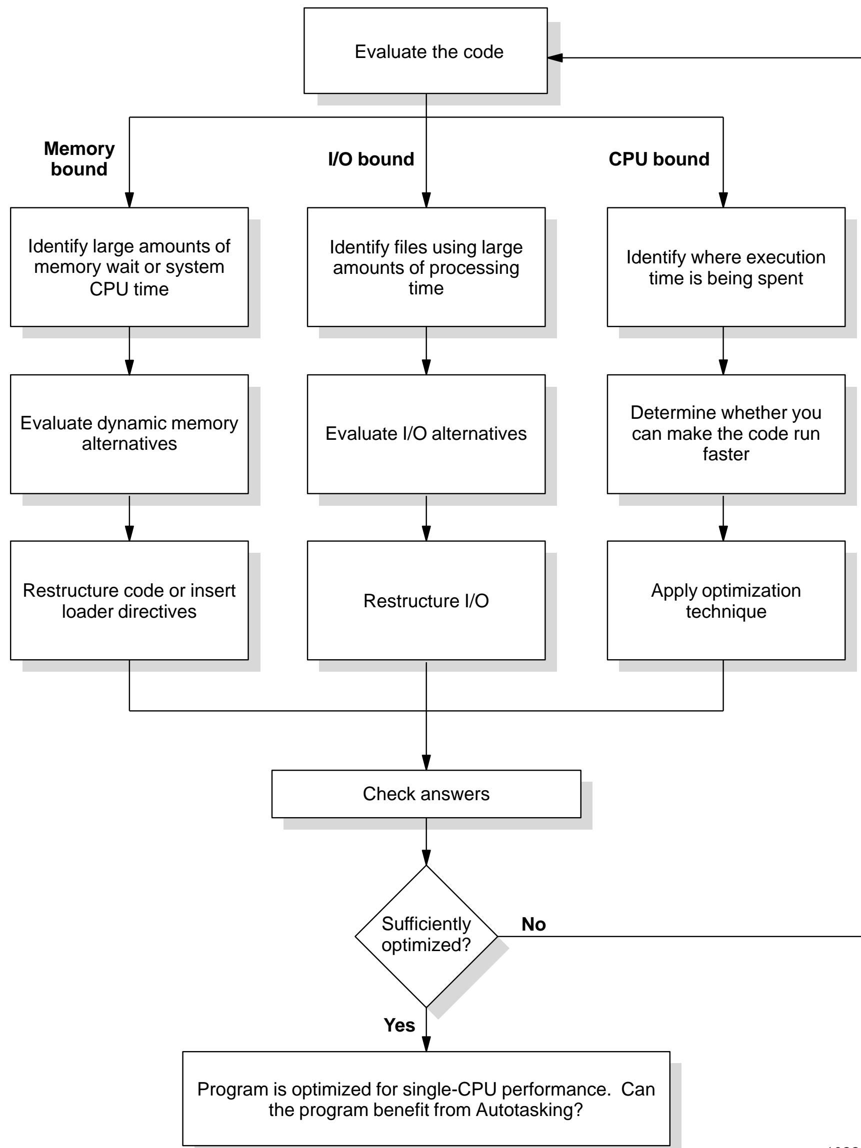


Figure 1. Optimization Overview

1.3 Hardware Overview

There are three models of the Cray SV1 series of systems:

- The Cray SV1 model, which is the system as originally released.
- The Cray SV1e model, which adds a faster processor to the original system.
- The Cray SV1ex model, which adds faster memory to the Cray SV1e model and supports the Solid State Storage Device (SSD-I). See Section 6.8, page 94 for more information regarding the SSD-I.

Because the Cray SV1 series system performs multistreaming, its processors are referred to as multistreaming processors (MSPs). Each MSP divides parallel work among four single-streaming processors (SSPs). See Chapter 3, page 25 for information on optimizing your program using multistreaming.

Each SSP has 2 add and 2 multiply functional units, allowing them to deliver 4 floating-point results per processor clock cycle.

The Cray SV1 series systems are significantly different from previous Cray vector machines in that they provide a cache for the data resulting from scalar, vector, and instruction buffer memory references. Like their predecessors, the Cray SV1 series systems achieve high bandwidth to memory for both unit and non-unit stride memory references.

The memory architecture is uniform access, shared central memory. Uniform memory access (UMA) means that the access time for any CPU memory reference to any location in memory is the same. Memory capacity for the system ranges from a minimum of 4 GB up to a maximum of 128 GB.

The following table describes differences among the models:

The Cray SV1 series system has two module types: processor and memory. The system must be configured with eight memory modules and one to eight processor modules. Each processor module has four CPUs.

Cray SV1 systems have Cray SV1 CPUs.

A Cray J90 processor module can be upgraded with a Cray SV1 processor module and the Cray J90 system can be configured with both processor module

types, Cray J90 or Cray SV1. The original Cray SV1 model processors cannot be mixed with Cray SV1e or Cray SV1ex model processors.

	Cray SV1	Cray SV1e	Cray SV1ex
Number of SSPs	4-32	4-32	4-32
Clock rate of an SSP in megahertz	300	500	500
Peak GFLOPs per SSP	1.2	2.0	2.0
Peak GFLOPs for the system	38.4	64	64
Memory size in GB	4-32	4-32	32–256*

* *Maximum value includes SSD-I.*

The following figure shows the block diagram for a single processor.

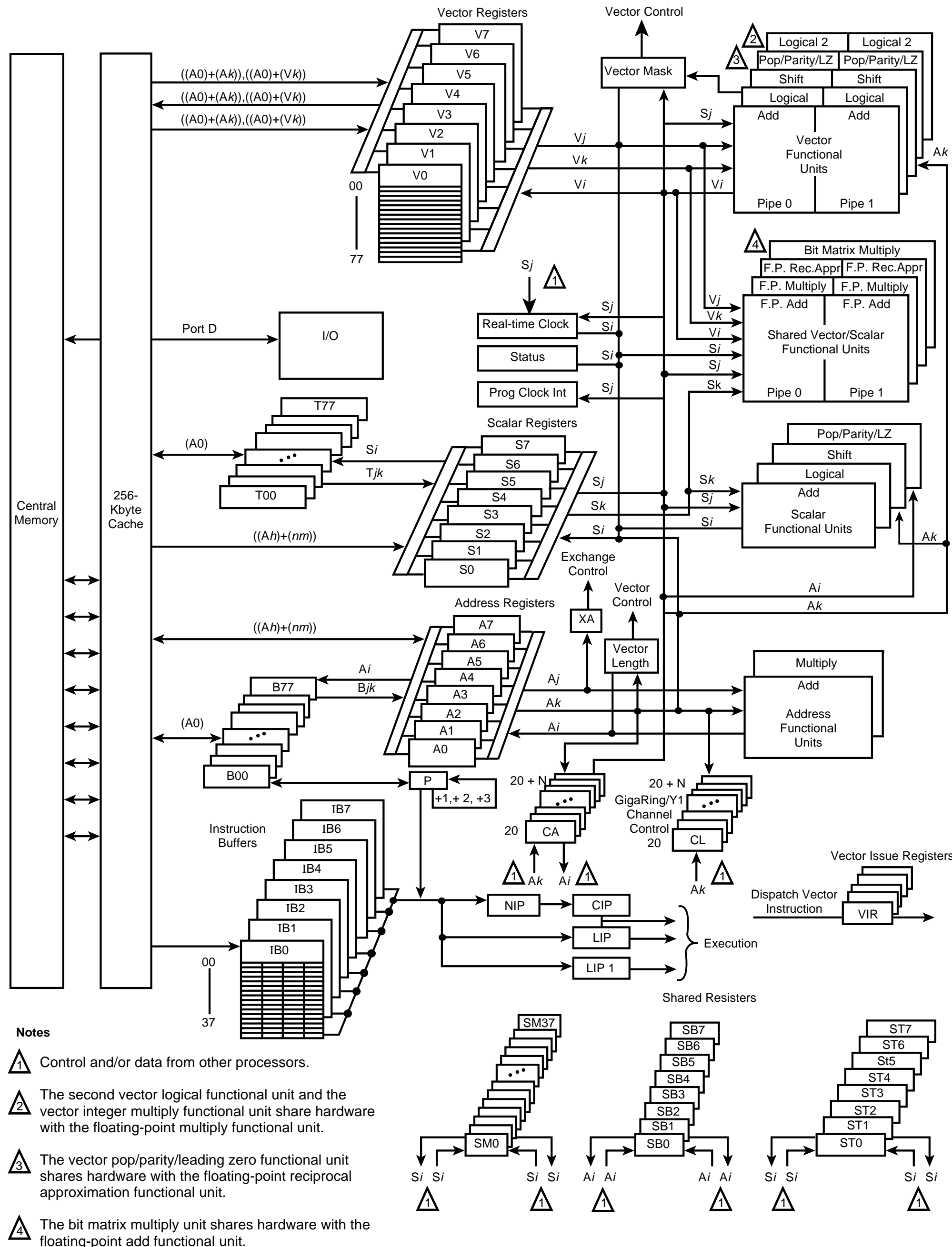


Figure 2. Block Diagram

1.3.1 The Processor

The Cray SV1 system has Cray SV1 processors. Cray SV1ex systems have Cray SV1e processors. (Some Cray SV1ex systems also have extended memory, called ex memory.)

The Cray SV1 processor uses a custom CMOS chip. The processor is implemented using two chip types: processor and cache.

The processor chip contains the vector and scalar units. Scalar registers, scalar functional units, and the instruction buffers reside in the scalar unit, while the vector unit contains vector registers and the vector functional units. As in previous Cray vector systems, the processor contains eight vector (V) registers, eight scalar (S) registers backed by 64 T registers, and eight address (A) registers backed up by 64 B registers. A parallel job also has access to eight shared B and eight shared T registers, which are used for low overhead data passing and synchronization between processors.

A vector functional unit contains two pipes, each capable of producing a result every processor clock cycle. This results in a peak rate for a functional unit of two results per clock cycle. The maximum vector length, or VL, is 64. The combined floating-point functional units, add and multiply, deliver four results per processor clock cycle.

In addition to the add and multiply units, the other vector functional units are reciprocal, integer add, shift, pop/parity/leading zero, bit matrix multiply, and logical.

The vector units are capable of full *chaining* and *tailgating*. Chaining is reading from a V register that is still being written to, and tailgating is writing to a V register that is still being read from a prior vector instruction. Scalar floating-point operations are executed using the vector functional units. This is different from the Cray J90 system, which has separate floating-point functional units for scalar operations.

Two data paths or ports are provided to move data between processor registers and memory via cache. In any given clock cycle, two memory requests can be active and consist of two dual-port reads or one dual-port read and one dual-port write. If there are no read requests, there can be only one write request active. The processor can access all of memory, but an application is limited to a 16-GB address space.

Instructions are decoded by the scalar unit; when vector instructions are encountered, they are dispatched to the vector unit, which maintains an independent instruction queue. Barring instruction dependency, the two units can execute independently.

There are 32 performance counters in four groups of eight each. Only one group can be active at a time, with software providing user access to the data. The groups are labeled 0 through 3. The collection order, based on how useful the performance information is to the user, is 0, 3, 2 and 1. For an example of using the hardware performance monitor, see Section 2.1.1, page 13.

Note: In addition to the processor clock, there is a system clock that runs at the rate of 100 MHz. When using the `cpu` instruction to return the count of clock ticks, the tick count is generated by the system clock rate.

1.3.2 Cache

Cache lies between memory and a processor's registers. Its purpose is to speed up loads.

Data read from memory is accessed through a high-speed, 32-Kword cache. Each of the processors involved in multistreaming has its own cache.

The Cray SV1 system cache is a four-way *set associative* temporary storage area, meaning each *line* can be allocated into any of four places, or *ways*.

Moving data from cache to a register is up to ten times faster than moving data directly from memory to a register. Having the data items you are going to use available in cache can represent a significant optimization in itself.

On non-cached Cray vector systems, performance on vector constructs generally increases as a predictable function of vector length. This is not always the case on the Cray SV1 series of systems, since long vectors can lead to a reduction in data cache efficiency. In general, it is better to use blocked algorithms (similar to those commonly used on microprocessors), balancing the vector length against any potential data reuse that can be exploited via data cache.

For information on optimizing cache, see Chapter 3, page 25.

1.3.3 I/O

Disk drives, interfaces to other networks, and other peripherals are connected to the Cray SV1 series of systems using the high-speed GigaRing I/O system. The double-ring product is illustrated in Figure 3, page 10.

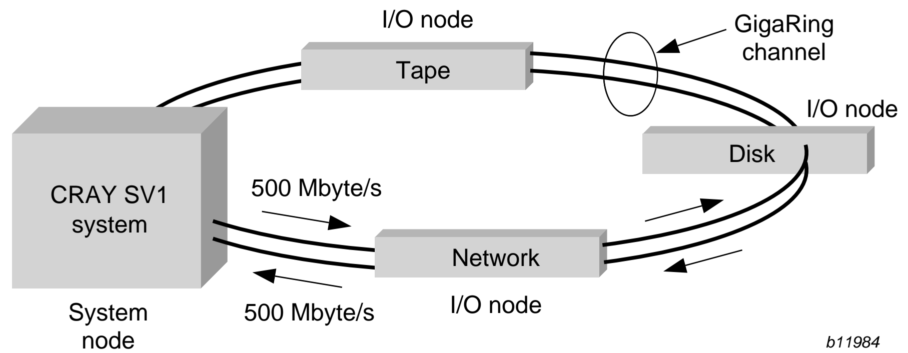


Figure 3. GigaRing I/O

Each of the rings has a maximum transfer rate of 500 MBPS, which provides an effective total bandwidth of 800 MBPS. Since the two rings rotate in different directions, the shortest path to the target node is selected for each transfer.

Evaluating Code [2]

The first step in optimizing a program is evaluating its overall performance. This allows you to decide where to focus optimization efforts.

When you compile and execute a program on a Cray SV1 series system, it usually will be dominated by one of the following general activities:

- Processor computation (Section 2.1, page 11)
- Memory management (Section 2.2, page 19)
- Input/output (I/O) processing (Section 2.3, page 24)

A program is considered to be memory bound, I/O bound, or CPU bound because its performance is limited by the dominant activity. Use the job accounting (`ja`) and hardware performance monitor (`hpm`) tools as directed in this chapter to identify where in your code to obtain the greatest potential performance gain.

Figure 4, page 12 summarizes the recommended initial analysis.

2.1 CPU-Bound Programs

A *CPU-bound code* spends most of its elapsed time performing calculations in the processor. (The terms *CPU* and *processor* are used in this section in the same sense as *SSP*, or single-streaming processor.) The `hpm`(1) report will tell you how effectively the code is using vector registers, instruction buffers, and memory ports.

The following sections provide information and steps for determining processor-related code bottlenecks. Section 2.1.1, page 13 describes the types of reports that you can generate with the `hpm`(1) command. Procedure 1, page 13 describes the use of the `hpm`(1) command to determine whether code is performing at peak levels on Cray SV1 series systems.

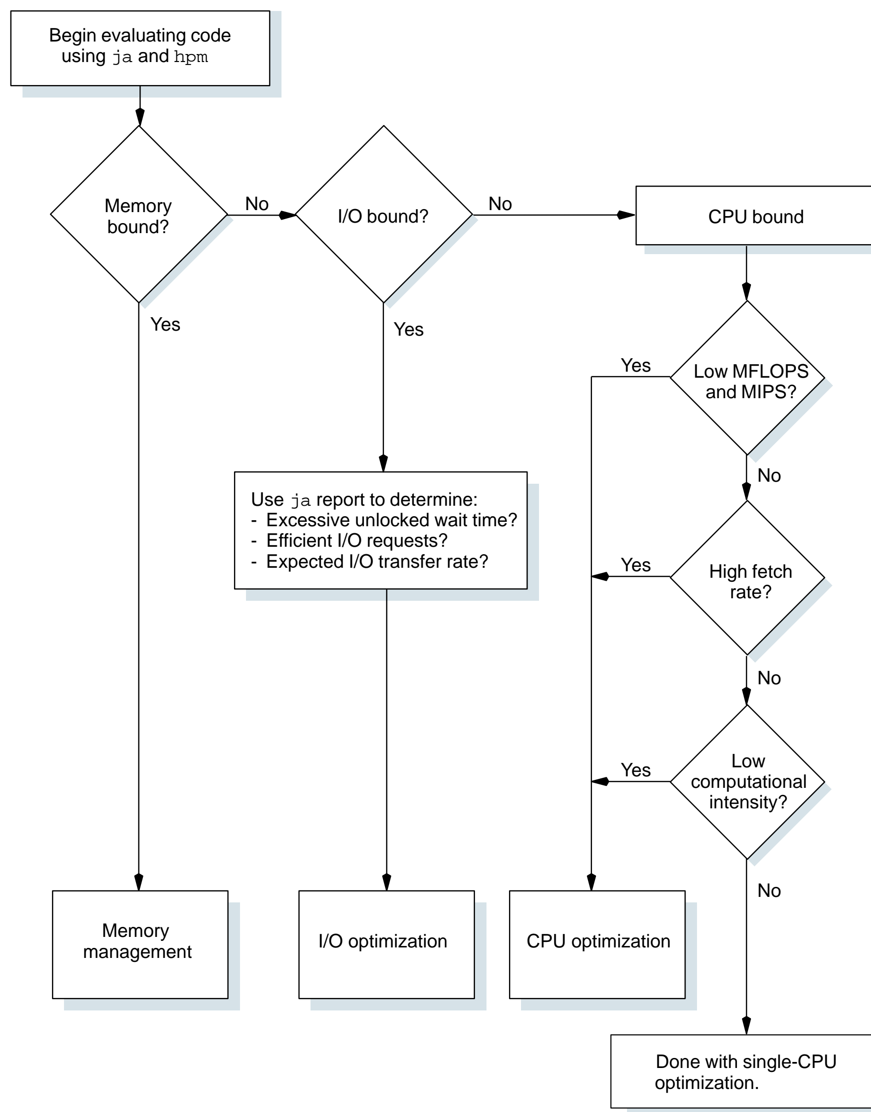


Figure 4. Evaluating Code

2.1.1 Using the `hpm` Command to Issue Reports

The `hpm`(1) command allows you to access the hardware performance monitor (HPM) and obtain overall program timing information. The `hpm` tool can issue any one of the following four types of reports:

- Program summary (HPM counter group 0)
- Hold-issue conditions (HPM counter group 1)
- Memory use (HPM counter group 2)
- Vectorization (HPM counter group 3)

For complete information on the `hpm` utility, see the `hpm`(1) man page or the *Guide to Parallel Vector Applications*.

Procedure 1: Using the `hpm` Command to Determine Processor Performance Statistics

Use the following procedure to access the code's processor performance statistics and to determine whether the code is optimized for single-CPU speed:

1. Produce an `hpm` report by first compiling the program, then issuing the `hpm` command with the program name, as shown in the following example:

   ```
   ftn yourcode.f   or   CC yourcode.C
   hpm ./a.out
   ```

   By default, this `hpm` command generates a program summary (group 0) report, the most commonly run report. Also, by default, the `hpm` command writes report-style output to the `stderr` file, allowing the program's default output to appear on the screen.

The following is a sample hpm group 0 report.

```
Group 0:  CPU seconds    :   31.74961      CP executing     :     9524883354

Million inst/sec (MIPS) :      42.72      Instructions     :     1356230656
Avg. clock periods/inst :       7.02
% CP holding issue      :      81.29      CP holding issue :     7743041828
Inst.buffer fetches/sec :       0.24M     Inst.buf. fetches:        7680895
Floating adds/sec       :      78.46M     F.P. adds        :     2491024131
Floating multiplies/sec :      82.85M     F.P. multiplies  :     2630592592
Floating reciprocal/sec :       0.00M     F.P. reciprocals :          10201
Cache hits/sec          :      92.75M     Cache hits       :     2944803625
CPU mem. references/sec :     139.73M     CPU references   :     4436364260

Floating ops/CPU second :     161.31M
```

2. Determine whether the code is dominated by scalar or vector operations. To do this, use the values shown in the hpm report to perform the following steps:

   a. Find the millions of instructions per second (MIPS) for the code by looking at the `Million inst/sec (MIPS)` row of the hpm report.

   b. Find the floating-point operations per second (FLOPS) for the code by looking at the `Floating ops/CPU second` row of the hpm report.

   c. Use Table 1, page 15 to determine whether the MIPS and FLOPS for the code are low or high, based on the hardware expectations.

      The code is not likely to achieve peak FLOPS or MIPS rates, but some codes are capable of performing at substantial fractions of peak speed.

      Low FLOPS is any single-digit rate up to 20% of peak rates for that system. High FLOPS are generally anything above 66% of peak rates for that system.

      Note: This assumes that the program uses floating point operations. In some applications, for example in the bioinformatics field, a fully vectorized and properly optimized program will have **zero** FLOPS.

      Low MIPS rates are at or near the bottom of the range for that system. High MIPS rates are generally anything near 50% of peak MIPS for that system.

   d. Use Table 2, page 15 to determine whether the code is dominated by scalar or vector operations.

Table 1. Single-Processor Hardware Expectations

	Cray SV1	Cray SV1ex
Peak FLOPS	1200M	2000M
MIPS range	30 to 300	50 to 500

Table 2. Determining if the code is dominated by scalar or vector operations

FLOPS	MIPS	Determination
High or Medium	Low	Code is most likely dominated by vector operations. The code is performing well.
Medium	Medium	Code is most likely a mix of scalar and vector operations. CPU optimization might help improve its CPU performance.
Medium or Low	High	Code is most likely dominated by scalar operations (is not vector code). CPU optimization will help improve its CPU performance.
Low	Medium	Code is dominated by scalar operations. CPU optimization will help improve its CPU performance.
Low	Low	Code is performing poorly, whether it is scalar or vector. It has a CPU performance problem. CPU optimization will help improve its CPU performance.

Note: Low FLOPS values are not always bad. In some applications, a fully vectorized and optimized program will use no floating-point operations, so it will have zero FLOPS.

3. Determine if the instruction buffer fetches per second (as shown by the `Inst.buffer fetches/sec` row of the `hpm` report) are close to or greater than 0.1 million per second. If yes, see the following note on analyzing the CPU-bound code. If no, go to the next step.

Note: Excessive instruction buffer fetches tend to slow down the code. If the code has a high rate (approaching 0.1 million per second), the code may have excessive jumping (as with `go to` or `if-then-else` constructs) or excessive calls to subprograms. CPU optimization probably will help the overall performance.

4. Determine the computational intensity ratio of the code. Use the values in the `hpm` report, as follows:

   a. Divide the number in the `Floating ops/CPU second` row by the number in the `CPU mem.references/sec` row. This is the computational intensity of the code.

      The computational intensity ratio is the ratio of the floating-point operation rate to the memory access rate. This ratio should reflect the floating-point operations in the code (for example, `a=b+c` has 1 `Floating ops/CPU second` and 3 `CPU mem.references/sec`, or a computational intensity of 0.33). Any ratio less than 1/3 makes excessive use of the processor memory ports while the remainder of the processor idles. This is usually caused by memory-to-memory traffic (`a=b`), highly scalar code, or a hidden performance problem.

   b. If the computational intensity ratio of the code is less than 1/3, the program is not making effective use of the processor. If the computational intensity ratio of the code is 1/3 or more, go to the next step.

5. If you have already determined that the code is not memory or I/O bound, the code is probably optimized for single-CPU performance. However, if your program still does not achieve the performance expected from your system after you have exhausted all optimization techniques, Cray can offer assistance for further optimization on a fee basis. See your Cray representative for more information.

6. If you have completed single-CPU optimization of your program and are satisfied with its single-CPU performance, you might want to run your program in parallel on multiple processors. The multistreaming processor (MSP) feature is the most convenient method to achieve parallel execution. To determine whether your program can benefit from multistreaming, refer to Chapter 3, page 25.

2.1.2 Analyzing Multistreaming Code

The following other tools are available to analyze multistreaming code:

- The `prof`(1) and `profview`(1) commands (see Section 3.6.1, page 33)
- The `MSP_STATS` environment variable (see Section 3.6.2, page 35)
- The `ps`(1) command (see Section 2.1.2.1, page 17)
- The `jstat`(1) command (see Section 2.1.2.2, page 18)
- The `cpu`(8) command (see Section 2.1.2.3, page 18)
- The `ja`(1) utility (see Section 2.2.2, page 20)
- The `sysconf`(1) command (see Section 2.2.2.1, page 22)

2.1.2.1 The `ps`(1) Command

The `-l` option of the `ps` command reports the process connection state in the `WCHAN` field of its output.

If you are running a single-processor job, an SSP process connected to CPU 5 would have `cpu-05` in the `WCHAN` field. When running on an MSP, `ps` displays both the MSP number and the number of each of the SSPs involved. The following convention applies:

`msp`*n.i*

n MSP number.

i SSP number.

For example, `ps -el | grep msptest`, while multistreaming the executable program `msptest`, produces output similar to the following:

```
 F S   UID   PID  PPID C PRI NI     ADDR      SZ       WCHAN TTY   TIME CMD
101 R  1330  4711  4689 0 996 24   250164    8504      msp0.2    p002  0:07 msptest
101 R  1330  4709  4689 0 996 24   250164    8504      msp0.0    p002  0:07 msptest
101 R  1330  4712  4689 0 996 24   250164    8504      msp0.1    p002  0:07 msptest
101 R  1330  4710  4689 0 995 24   250164    8504      msp0.3    p002  0:07 msptest
```

Output for a multitasking and multistreaming program running on two MSPs is as follows:

```
 F S   UID   PID  PPID C PRI NI     ADDR      SZ       WCHAN TTY   TIME CMD
101 R     0  2246  2203 0 995 24   264064   16504      msp0.1    p003  0:17 msptest
101 R     0  2245  2203 0 996 24   264064   16504      msp0.0    p003  0:17 msptest
101 R     0  2247  2203 0 995 24   264064   16504      msp0.2    p003  0:17 msptest
101 R     0  2248  2203 0 995 24   264064   16504      msp0.3    p003  0:17 msptest
101 R     0  2249  2203 0 996 24   264064   16504      msp1.0    p003  0:17 msptest
```

```
101 R     0  2250  2203 0 996 24   264064   16504     msp1.1    p003  0:17 msptest
101 R     0  2251  2203 0 996 24   264064   16504     msp1.2    p003  0:17 msptest
101 R     0  2252  2203 0 993 24   264064   16504     msp1.3    p003  0:17 msptest
```

There is no ordering imposed on the output.

2.1.2.2 The jstat(1) Command

The `jstat` command displays information pertaining to active jobs. Using the `-j` option, you can view the processor connection state of all processes within a job. The `state` field of the `jstat -j` *id* output identifies the MSP and SSP on which each executing process within the job is running. For example:

```
pid    status   state    utime   stime    size    addr   system call   command
 6647  running  msp0.1     117       0    16504   90116  nosys         msptest
 6652  running  msp0.0     117       0    16504   90116  nosys         msptest
 6651  running  msp0.2     117       0    16504   90116  nosys         msptest
 6649  running  msp0.3     117       0    16504   90116  nosys         msptest
```

The order of the SSPs is random.

2.1.2.3 The cpu(8) Command

The `-i` option of the `/etc/cpu` command displays information for each configured processor. Among the items displayed is an indicator identifying participation in an MSP. The following example shows output for a 12 CPU machine with one MSP configured:

```
cpu  type  state
  0   sv1  up    cache defon
  1   sv1  up    cache defon
  2   sv1  up    cache defon
  3   sv1  up    cache defon msp0.2
  4   sv1  up    cache defon
  5   sv1  up    cache defon
  6   sv1  up    cache defon
  7   sv1  up    cache defon msp0.0
  8   sv1  up    cache defon
  9   sv1  up    cache defon
 10   sv1  up    cache defon msp0.3
 11   sv1  up    cache defon msp0.1
```

2.2 Memory and Cache

There are two ways of optimizing memory on the Cray SV1 system:

- By getting the most out of *cache*. Cache is a high-speed memory located between main memory and each MSP. The time it takes a register to access data from cache is four to five times faster on the Cray SV1 system and 10 times faster on the Cray SV1ex system than the time it takes to access data from memory. See Section 2.2.1, page 19.
- By minimizing the size of your program in memory. When the compiled program begins execution, it becomes a UNICOS user process in memory. The process resides in memory during execution and it interacts with the operating system. Its size might expand or decrease during execution. If the process spends excessive elapsed time managing memory, it is considered a *memory-bound process*. See Section 2.2.2, page 20.

2.2.1 Cache

For information on cache hits and cache hits per second, see the group 0 `hpm` example in Section 2.1.1, page 13. Use the group 2 report for the hit rate for cache reads as follows:

```
% ftn myprog.f
% hpm -g2 ./a.out >hpm.out
```

The contents of the `hpm.out` file is as follows.

```
Group 2:  CPU seconds   :       30.82432     CP executing  :      9247297374

Inst. buffer fetches/sec   :       0.25M  total fetches    :         7675226
Scalar memory refs/sec     :       0.74M  actual refs      :        22724168
    % of all data refs:    0.51%
  Scalar memory writes/sec :       0.51M  actual writes    :        15683763
  Scalar memory reads/sec  :       0.23M  actual reads     :         7040405
    cache read hit rate:  52.14%          actual hits      :      1525073379
Block memory refs/sec      :     142.94M  actual refs      :      4406125465
  B,T memory refs/sec      :       2.04M  actual refs      :        62784808
  V memory refs/sec        :     140.91M  actual refs      :      4343340657
CPU memory refs/sec        :     143.68M  actual refs      :      4428849633
    avg conflict/ref:   0.05              actual conflicts :       211824843
  CPU memory writes/sec    :      46.76M  actual refs      :      1441268906
  CPU memory reads/sec     :      96.92M  actual refs      :      2987580727
```

For information on how to increase the hit rate and improve the performance of your program, see Section 5.2, page 62.

2.2.2 Code Size

A general rule for the size of a code's process in memory is "smaller is better." The advantages of a smaller process are reductions in swap frequency, wait-time, and time-to-load. Although there are exceptions to this rule, for now you need only to identify whether your process is memory bound.

The `ja` utility reports job or session-related accounting information provided by the job accounting daemon. It can provide you with a report that contains an overall view of memory usage, I/O, and system overhead. This report can help you determine whether your code is CPU bound, memory bound, or I/O bound.

For more information on the job accounting tool, see the `ja`(1) man page.

Procedure 2: Determining if the code is memory bound

To determine if the code is memory bound, perform the following steps:

1. To obtain a report that shows an overall view of memory usage, I/O, and system overhead, run the job accounting utility, `ja -clth` (single-tasked), with the code, as shown in the following example. (To save an extra step, you can also run `ja`(1) concurrently with `hpm`(1), the results of which you will use later).

   Enter this command to compile and load the code:

   ```
   ftn -o bigio bigio.f
   ```

   Enter this command to turn on job accounting:

   ```
   ja
   ```

   Enter this command to execute the code as a UNICOS user process:

   ```
   ./bigio
   ```

   Enter this command to store a 132-column report in the `ja.rpt` file:

   ```
   ja -clth > ja.rpt
   ```

Here are some samples of requests for and returns from sample `ja` reports:

```
] cut -c1-72 ja.rpt

Job Accounting - Command Report
===============================

Command  Started    Elapsed    User CPU   Sys CPU   I/O Wait I/O Wait
  Name      At      Seconds     Seconds   Seconds   Sec Lck  Sec Unlck
======== ======== =========== ========== ========== ======== ==========
ja       11:29:50      0.1906     0.0014     0.0137     0.18     0.0000
csh      11:29:59      0.0038     0.0004     0.0035     0.00     0.0000
bigio    11:30:05     31.1637    31.0011     0.0350     0.13     0.0000


] cut -c1-9,73-132 ja.rpt

Job Accou
=========

Command   CPU MEM   I/O WMem Kwords   Log I/O Memory   Ex
  Name    Avg Mwds  Avg Mwds Xferred  Request HiWater  St Ni Fl  SBU'
======== ========= ========= ======== ======= ======= === == == =====
ja          0.3556    0.4297     0.00       0     880   0 20        0.
csh         0.0734    0.0000     0.00       2     163   0 20 F      0.
bigio       0.7890    0.7793     0.14      28    1616   0 20        0.
```

Note: To ensure a representative reading, the user process you are measuring with the `ja` utility should have an execution time that is greater than 1 second.

2. Find the `Memory HiWater` column of the `ja` report. Perform the following steps to determine if the memory high-water mark for the process is a significant fraction of available user memory.

   Note: A significant fraction of available user memory on your system is a subjective measure based on the chances of the code finding room in memory as it competes with other users' jobs. Whether the code finds room in memory depends on how many jobs are competing for memory, the job size, the priority of the process, memory latency, and other factors.

   a. Multiply the number in the `Memory HiWater` column of the `ja` report by 512. This gives you the maximum size of your process as measured in Cray words.

b. Determine the amount of available user memory and the number of available processors on your system by using the `sysconf`(1) command (see Section 2.2.2.1, page 22). Use the `USRMEM` value in the `SOFTWARE` report of the `sysconf` command as the amount of available user memory for your system. Use the `NCPU` value in the `HARDWARE` report of the `sysconf` command as the number of available processors.

c. Although you might be able to execute a process as large as the size stated in `USRMEM` of the `sysconf` report, you are probably sharing your system with other users. Also, your system most likely has multiple processors. Therefore, use the following guidelines to determine whether the memory high-water mark for the process is a significant fraction of available user memory.

- If the system on which the code is running has four processors or less, divide the maximum size of your process by user memory (`USRMEM` on the `sysconf` output). If the result is equal to or greater than 0.333, your process will probably benefit from optimization of memory management. If the result is less than 0.333, go to the next step.

- If the system on which the code is running has more than four processors, divide the user memory (`USRMEM` on the `sysconf` output) by the number of processors (`NCPU` on the `sysconf` output). Compare this number to the maximum size of your process. If the maximum size of your process is bigger than this number, your process will probably benefit from optimization of memory management. If the maximum size of your process is not bigger than this number, go to the next step.

3. If the number in the `Sys CPU Seconds` column in the `ja` report for the code is greater than 10% of the number in the `User CPU Seconds` column, you have memory-bound code. Inefficient memory management is one cause for excessive elapsed time.

   If the number in the `Sys CPU Seconds` column is not greater than 10% of the number in the `User CPU Seconds` column, your code is not memory bound.

2.2.2.1 Determining How Much Memory Is Available on Your System

To find out how much user memory is available on your Cray SV1 series system, use the `sysconf`, `target`, or `sar -M` commands, or ask your system administrator.

The sysconf(1) command reports a number for user memory (USRMEM) in the SOFTWARE portion of its report, as shown in the following sample portion of a sysconf command output.

```
HARDWARE: SERIAL= SN9617   MFTYPE= Cray SV1   MFSUBTYPE= SV1
          NCPU= 20   NSSP= 8   NMSP= 3   CPCYCLE= 10.0000 ns
          MEM= 4294964992   NBANKS= 1024   CHIPSZ= 67108864
          AVL= YES   BDM= YES   EMA= YES   HPM= YES   BMM= YES
          SSD= 0   SSDRINGS= 0   IOS= MODEL_F   RINGS= 4

SOFTWARE: RELEASE= 10.00   POSIX VERSION= 199009   SECURE SYS= ON
          SYSMEM= 60092416 WRDS   USRMEM= 4234872576 WRDS
          OS_HZ= 60   CLK_TCK= 100000000
          JOB_CONTROL= YES   SAVED_IDS= YES   SCTRACE= ON
          UID_MAX= 16777215   PID_MAX= 100000
          ARG_MAX= 49999   CHILD_MAX= 98   OPEN_MAX= 64
          NMOUNT= 150   NUSERS= 300   NPTY= 200
          NDISK= 32    SDS= 0    NBUF= 8000
          POSIX_PRIV= ON   SECURE_MLSDIR= SECURE   SECURE_MAC= OFF
          PRIV_SU= ON    PRIV_TFM= OFF
```

The target(1) command reports the absolute memory size of your system in the memsize value as shown in the following sample portion of a target command output. Available user memory should be greater than 90% of this value.

```
Primary machine type is: CRAY SV1
   banks    = 1024
   numcpus  = 20
   ibufsize = 32
   memsize  = 4294967296
   memspeed = 33
   clocktim = 10000
   numclstr = 21
   bankbusy = 14
```

The sar(8) command with the -M option specified provides a dynamic report of available user memory.

For more information on the sysconf(1), target(1), and sar(8) commands, see the man pages.

2.3 I/O Bound

If a program spends most of its elapsed time performing I/O, it is considered *I/O bound*. I/O optimization can offer a significant savings in elapsed time. If the design of a program requires large amounts of I/O, you should optimize for I/O performance. However, if the code runs 24 hours and performs only 1 hour of I/O, there are other areas of optimization that will probably have a greater impact on overall code performance.

Procedure 3: Determining if the Code is I/O Bound

To determine whether the code is I/O bound, perform the following steps:

1. Run the job accounting utility `ja -clth` (single-tasked), with the code to get a report for an overall view of memory usage, I/O, and system overhead. You can use the same report you created in Procedure 2, page 20 without running the code again.

   Note: To ensure a representative reading, the user process you are measuring with the `ja` utility should have an execution time that is greater than 1 second.

2. If the sum of the number in the `I/O Wait Sec Lck` column plus the number in the `I/O Wait Sec Unlck` column of the `ja` report is close to or greater than 50% of the number in the `User CPU-Seconds` column, the code is probably I/O bound.

   Note: If a program has frequent or large formatted I/O requests, it is possible for it to be I/O bound even if the test in this step indicates that it is not. Formatted I/O (specified by using a Fortran `FORMAT` statement or a C++ `scanf` or `printf` command) consumes user CPU seconds to translate between ASCII and internal binary representation. This form of I/O time is not reflected in the two `I/O Wait` time columns of the `ja` report. A program that contains frequent or large formatted I/O requests might benefit from I/O optimization.

Multistreaming [3]

Multistreaming is a feature that lets you schedule four dedicated Cray SV1 series CPUs as one multistreaming processor (MSP) for a Fortran, C, or C++ program. It automatically divides loop iterations among the four CPUs, each of which is called a single streaming processor (SSP). You may get speedup factors of up to four on loops to which this technique can be applied.

Figure 5, page 25 illustrates the loop iterations that each CPU will operate on for the following loop.

```
DO I = 1,20
    A(I) = A(I) * 3.14
ENDDO
```

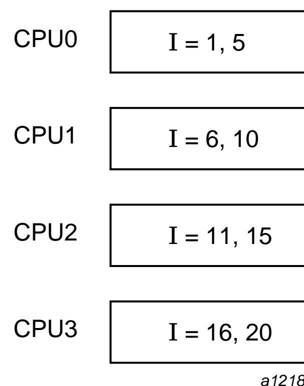


Figure 5. Dividing Loop Iterations among CPUs

Multistreaming is an optional feature. To determine whether or not multistreaming is enabled on your Cray SV1 series system, enter the `sysconf`(1) command at your system prompt. The `HARDWARE` output field contains the `NMSP` field, which shows the number of multistreaming processors configured. The relevant field is in boldface type in the following example of `sysconf` output. Having the `NMSP` field set to 2, as in this example, means two MSPs (eight SSPs) are configured for multistreaming.

```
HARDWARE: SERIAL= SN3202   MFTYPE= CRAY-SV1   MFSUBTYPE= SV14XX
          NCPU= 32   NSSP= 24   NMSP= 2   CPCYCLE= 10.0000 ns
          MEM= 1073739520   NBANKS= 1024   CHIPSZ= 16777216
```

```
            AVL= YES   BDM= YES   EMA= YES   HPM= YES   BMM= YES
            SSD= 0   RINGS= 0   IOS= MODEL_F

  SOFTWARE: RELEASE= 10.00   POSIX VERSION= 199009   SECURE SYS= ON
            SYSMEM= 33972224 WRDS   USRMEM= 1039767296 WRDS
            OS_HZ= 60   CLK_TCK= 100000000
            JOB_CONTROL= YES   SAVED_IDS= YES   SCTRACE= ON
            UID_MAX= 16777215   PID_MAX= 100000
            ARG_MAX= 49999   CHILD_MAX= 98   OPEN_MAX= 64
            NMOUNT= 150   NUSERS= 300   NPTY= 200
            NDISK= 32    SDS= 0    NBUF= 8000
            POSIX_PRIV= ON    SECURE_MLSDIR= SECURE    SECURE_MAC= OFF
            PRIV_SU= ON    PRIV_TFM= OFF
```

Multistreaming on a Cray SV1 series system is similar to Autotasking in distributing loop iterations across processors. However, multistreaming causes gang scheduling of all requested processors, meaning they are attached to the program whether they are actually executing code or not. Autotasking schedules processors as they are needed. Processor utilization efficiency is directly proportional to the extent that you are able to multistream the program. For information on how to invoke and tune your program to enhance multistreaming, see the following section.

3.1 Compiler Options and Directives

You can control multistreaming within your program by means of command-line options and compiler directives.

3.1.1 Fortran Compiler Options

To enable multistreaming in a Fortran program, include either the `-O stream1`, `-O stream2`, or `-O stream3` option on the `ftn`(1) command line. The `-O stream2` and `-O stream3` options are more aggressive than `-O stream1`. For more information on using these options with Fortran, see the *Cray Fortran Compiler Commands and Directives Reference Manual*.

In the following example, the `ftn`(1) command turns on streaming:

```
% setenv NCPUS 1
% ftn -O stream2 -O nothreshold streamer.f
% ./a.out
```

The `setenv` command specifies one multistreaming processor, meaning your program gets four SSPs.

> **Note:** You may get slight numeric differences when you select the most aggressive streaming option (`stream3`). Check your program output to see if the answers generated are acceptable.

3.1.2 C and C++ Compiler Options

In C and C++, use the `-h stream1`, `-h stream2`, and `-h stream3` compiler options to enable multistreaming. The `-h stream2` and `-h stream3` options are more aggressive than `-h stream1`. See the *Cray Standard C and Cray C++ Reference Manual* for more information on these options.

The following example specifies conservative streaming. By default, the program runs on one MSP. The `-h report=m` option generates multistreaming optimization messages.

```
% setenv NCPUS 1
% cc -h stream1 -h report=m -h nothreshold streams.c
% ./a.out
```

3.1.3 Directives

The `concurrent`, `preferstream`, `stream`, and `nostream` directives allow you to enhance the performance of multistreaming.

3.1.3.1 `concurrent` Directive

The `concurrent` directive lets you communicate to the compiler that a loop is parallel when the compiler cannot make that determination on its own. Placing the following in front of a loop instructs the compiler to multistream that loop.

Fortran:

```
!DIR$ concurrent
```

C and C++:

```
#pragma _CRI concurrent
```

This directive is especially important for C programs in which the presence of pointers can make parallelizing loops difficult for the compiler.

The `ivdep` directive serves this purpose for vectorization, but it is not a powerful enough assertion to allow the compiler to multistream a loop. Converting all `ivdeps` to `concurrent` in your program both enables multistreaming and improves the performance of vector code.

Applying the `concurrent` directive to all the loops that can be multitasked gives the compiler the best chance of generating optimal code. For information on the kinds of loops that can be multistreamed, see Section 3.2, page 29.

3.1.3.2 `preferstream, stream`, and `nostream` Directives

You can designate or skip loops using the multistreaming directives. The Fortran directives are as follows:

```
!DIR$ preferstream
!DIR$ nostream
!DIR$ stream
```

The `preferstream` directive selects which loop, among two or more nested loops, to be multistreamed. Insert the directive immediately in front of the loop to be multistreamed. If the compiler has determined that it is safe to multistream the loop, it will do so.

The `nostream` and `stream` directives toggle multistreaming off and on in Fortran. The `nostream` directive turns multistreaming off in your program until a `stream` directive is encountered.

The following Fortran example turns streaming on for the inner loop:

```
     DO I=1,N1
!DIR$ preferstream
          DO J=1,N2
            A(I,J) = B(J,I)
          ENDDO
       ENDDO
```

The C and C++ directives are as follows:

```
#pragma _CRI nostream
#pragma _CRI preferstream
```

Unlike Fortran, there is no `#pragma_CRI stream` directive. The `nostream` directive has different meanings in Fortran and C. The C and C++ version turns multistreaming off only for the loop that immediately follows the directive.

In cases in which the compiler could multistream more than one loop in a loop nest, the `preferstream` directive instructs it to choose the one immediately following the directive.

3.2 Loops That Are Multistreamed by the Compiler

The compiler uses certain criteria to judge whether or not a loop can be automatically multistreamed.

At the point the compiler evaluates a program, the code has already gone through an initial restructuring, as it would for vectorization or multitasking. Optimizations such as loop interchange (switching an inner loop with an outer loop) and loop splitting (changing a single loop into two or more loops) may have been done.

If a loop passes the following tests, it will be multistreamed:

- Its iterations can be divided among different processors without delivering incorrect results. The loop can contain private arrays if their values are not used outside the loop.
- There are no function or subroutine calls within the loop. (The Autotasking `CNCALL` directive is ignored.)
- A scatter operation is ordered as opposed to random. That is, there must be a constant stride through the array to be scattered; taking random elements from an array cannot be multistreamed.
- The loop is not contained in a vector loop.
- It is not prefaced by the `nostream` directive.
- It has a trip count of at least two.
- While the `-O threshold` Fortran option or the `-h threshold` C and C++ option does not disable multistreaming, it does slow down a program's execution. Because `threshold` is the default, you must specify `-O nothreshold` or `-h nothreshold` to turn it off.

When two loops that qualify to be multistreamed are nested, the following criteria are used in the following order to choose which loop to multistream:

1. The outermost loop that is prefaced by the `preferstream` directive.
2. The loop that the compiler estimates will have the greatest amount of work after the loop is interchanged to its outermost valid position.

3. The outermost loop after initial restructuring.

Once a loop has been selected, it is moved to its outermost valid position before multistreaming. If the loop is not prefaced by a `preferstream` directive and has a runtime trip count, two versions of the loop are generated, one multistreamed and the other not. Which version is executed at runtime depends on the trip count.

3.3 Bit Matrix Multiply (BMM) and Multistreaming

Your view of the BMM programming model on a Cray SV1 series system is identical to that used on all previous Cray platforms: a single BMM register, 64 elements deep and wide, and the intrinsic functions. The Fortran functions are `m@ld`, `m@mx`, `m@ldmx`, `m@ul`, and `m@clr`, and the C and C++ functions are `_mclr_mld`, `_mmx`, `_mldmx`, and `_mul`.

The compiler automatically utilizes the multiple BMM registers available on an MSP and divides the BMM work between them.

There are two types of BMM loops: those that *directly* contain BMM operations and those that *indirectly* contain BMM operations (that is, those that contain direct or indirect BMM loops). In the following loops, the `i` loops directly contain the `m@ld` and `m@mx`, and the `j` loop indirectly contains the operations:

```
do j=1, n
    do i=1, 64
      m@bmm(i) = m@ld( b(i,j) )
    enddo
    do i=1, m
      a(i,j) = m@mx( (c(i,j) )
    enddo
enddo
```

The following list uses these terms in describing how the compiler applies BMM operations to the SSPs of an MSP:

- Loops that directly contain an `m@ld` or `_mld` that is used outside the loop need to be executed redundantly across all SSPs. Loops that indirectly contain an `m@ld` or `_mld` can and should be multistreamed as usual.
- Loops that directly contain `m@ldmx` or `_mldmx` can be multistreamed, but partitions (except the last) must be multiples of 64. Loops that indirectly

contain m@ldmx or _mldmx can be multistreamed with no restrictions on partition size.

- Loops that contain m@mx or _mmx (directly or indirectly) can be multistreamed with no restrictions on partition size.
- Loops that directly contain m@ul or _mul cannot be multistreamed. They can, however, be executed redundantly across SSPs when containing a feeding m@ld or _mld, as in the following loop:

```
CDIR$ shortloop
              do i=0, vl-1
                m@bmm(i) = m@ld( m@ul() )
              enddo
```

- Loops that contain m@mx or _mmx with a reaching m@ld or _mld (or a m@ldmxor _mldmx) outside the loop and imported from a loop that is not redundantly multistreamed, first broadcast the BMM value from SSP 0 to the others before multistreaming. (All loops will be multistreamed so that SSP 0 will execute the last iteration of the loop. Consequently, SSP 0's BMM register will always contain the value that is indicated by the BMM programming model to be in the user's BMM register at the end of a loop.)
- Loops that directly or indirectly contain m@clr or _mclr can be multistreamed. If an m@clr or _mclr is not in a multistreamed loop, it should be executed redundantly across all SSPs.

3.4 Tasking and Multistreaming

Multistreaming and multitasking can coexist in the same program, as described in this section. Multistreaming offers the following performance advantages over multitasking:

- The startup time for multistreaming is faster.
- Communication between multistreaming processors is more efficient than between multitasking processors.

Still, some programs may run as fast or even faster when they use multitasking. Running your program both ways will tell you which method is better for you.

3.4.1 Multitasking on MSPs with Multistreaming

Multitasking and multistreaming can be combined within an application to provide better performance than either model alone. Within a function, the compiler will do either multistreaming or multitasking, but not both. If the function contains multitasking directives, the compiler will do multitasking; otherwise, it will do multistreaming.

Usually, the best approach to combining the models is to apply multitasking through user directives to parallel loops that contain subroutine calls, and to apply multistreaming within the called subroutines.

If you include both multitasking and multistreaming options on the compiler command line (for instance, `-O task2, -O stream2` in Fortran and `-h task2, -h stream2` in C and C++), the value of the `NCPUS` environment variable (discussed inSection 3.1, page 26) will refer to MSPs. If you set `NCPUS` to 2 before compiling your program, you will have two MSPs, or a total of eight SSPs, to work with.

In the following example, the loop will be multitasked by the compiler, and the iterations of the loop will be split between the two MSPs requested by the programmer:

```
!CMIC$ DOALL PRIVATE (I), SHARED (A,B,C,N)
       DO I = 1, N
         CALL FOO(A,B,C,I)
       ENDDO
```

The best processor utilization is achieved if the work within a call to `FOO` is effectively distributed through multistreaming across the SSPs in the MSP.

3.4.2 Multitasking on MSPs without Multistreaming

If you enable multitasking on the compiler command line (for instance, `-O task3` or `-h task3`) and do not specify multistreaming, the compiler will not multistream for the compilation. The value of the `NCPUS` environment variable (discussed inSection 3.1, page 26) now refers to SSPs instead of MSPs.

You may see improved performance for a multitasking program on Cray SV1 series systems without making changes to the code. A multitasking program will compete with multistreaming programs for processors if you set the `MP_DEDICATED` environment variable and enter the `cpu`(8) command as in the following example. `MP_DEDICATED` relieves some of the overhead of task scheduling. Notice that you must set the `NCPUS` environment variable to four times the value of the `cpu -a` argument.

```
% setenv NCPUS 8
% setenv MP_DEDICATED 1
% cpu -a 2 ftn -O task3 atasker.f
```

3.5 Vectorization and Multistreaming

Once multistreaming divides the iterations of a loop among the SSPs working on your program, a number of single-processor optimizations created by the compiler continue to improve the performance of your program. The most important is vectorization.

Vectorization usually yields a greater speedup than any of the other single-processor optimizations. The more you can get out of vectorization, the closer you can come to realizing the full potential of your Cray SV1 series system.

Vectorization and multistreaming coexist automatically, as follows:

- Vectorization and multistream can both work on the same loop, causing an MSP to behave like an 8-pipe vector.
- Multistreaming can be done on a loop outside of the vector loop, making it more similar to tasking.

For more information on vectorization, see Chapter 4, page 39.

The major strategies you can follow in order to enhance vectorization are:

- Keep the stride through the array small; ideally, use a stride of 1.
- Avoid less efficient vector code, such as loops that contain variant `IF`s and reductions.

3.6 Analyzing the Performance of a Multistreaming Program

Use the `prof`(1) command and `MSP_STATS` environment variable for analyzing a multistreaming program.

3.6.1 The `prof` and `profview` Commands

The `prof`(1) command returns timing information at the function and subroutine level for each processor involved in a program. To generate and view performance data for a Fortran program, use the `prof` library and command and the `profview`(1) visualization tool, as follows:

```
% ftn -l prof -O stream2 -V -v streamer.f
% setenv PROF_WPB 1
% ./a.out
% prof -x a.out >prof.report
% profview prof.report &
```

For a C or C++ program, use the following commands:

```
% cc -l prof -h stream2 -V myprog.c
% setenv PROF_WPB 1
% ./a.out
% prof -x a.out >prof.report
% profview prof.report &
```

Figure 6, page 35 is an example of a pie chart view generated by `profview`. This represents an aggregation of data collected from all four MSPs involved in executing the program. It shows you the percentage of time consumed by the longest-running routines.

When a program is multistreamed but not multitasked, the amount of time spent in barriers will depend on the percentage of the program that can be multistreamed. When the program is executing in multistreaming mode but is not currently processing a loop, only one SSP is executing the program; the others are waiting at a barrier until the next loop. This example shows that wait time in the `_MsBarrier` routine.

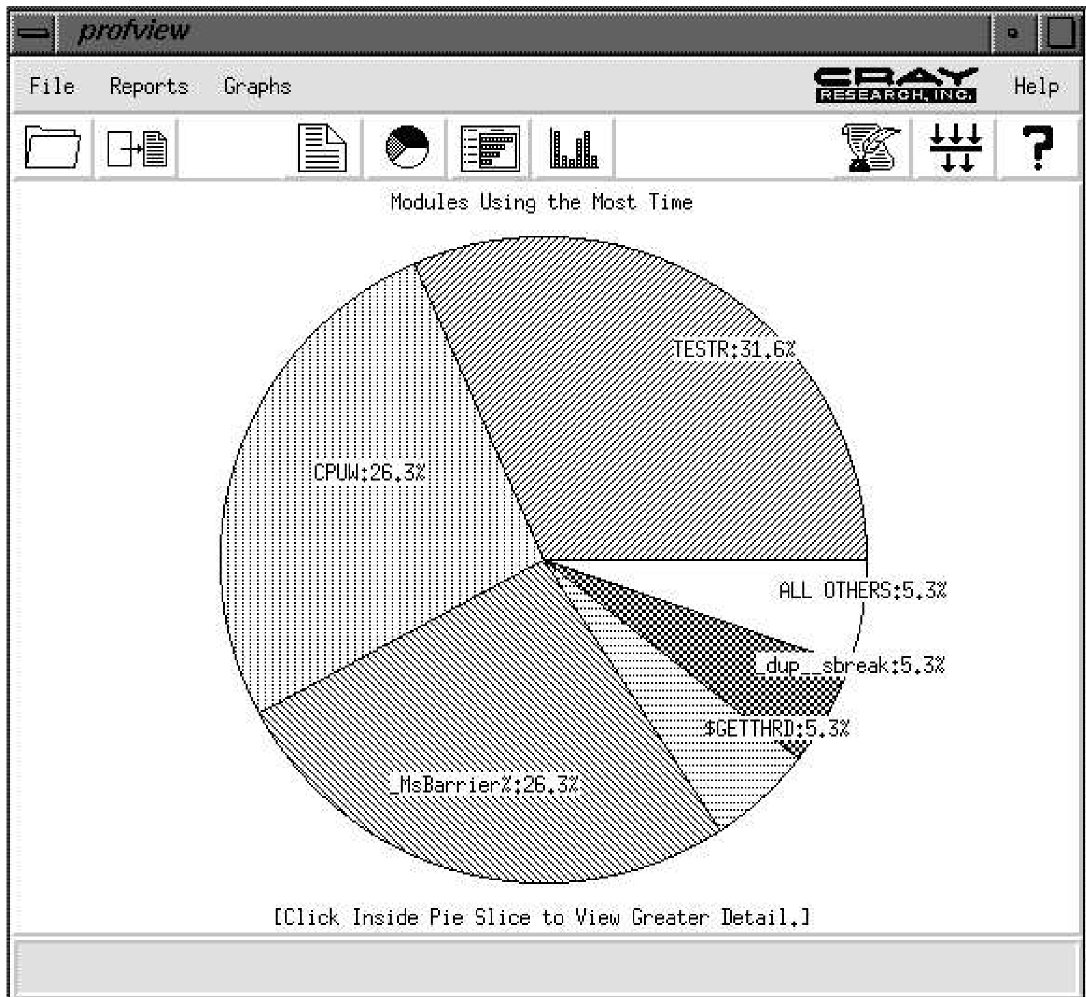


Figure 6. Profview Pie Chart

3.6.2 The `MSP_STATS` Environment Variable

The `MSP_STATS` environment variable provides statistics on multistreaming for Fortran, C, and C++ programs. To enable the generation of statistics, set `MSP_STATS` as follows:

```
% setenv MSP_STATS 1
```

Enter the following commands to run your Fortran program with aggressive multistreaming, on one multistreaming processor, and with MSP_STATS enabled:

```
% setenv MSP_STATS 1
% setenv NCPUS 1
% setenv NEW_ENTRY
% ftn -O stream2 -l libmsx.a myprog.f
% ./a.out
```

Running MSP_STATS for a C or C++ program involves the same set of commands:

```
% setenv MSP_STATS 1
% setenv NCPUS 1
% setenv NEW_ENTRY
% cc -h stream2 -l libmsx.a myprog.c
% ./a.out
```

MSP_STATS provides the following output:

```
MSP SSP    UserSecs   MsLibSecs   #MsEnts    #Parks    #Barrs #CInvs
--- --- ----------- ----------- --------- --------- --------- ---------
  0   0      35.612       2.029    777737    783438      5401 788839

  0   1       0.826      36.815    777737    783438      5401 788839

  0   2       0.826      36.815    777737    783438      5401 788839

  0   3       0.856      36.785    777737    783438      5401 788839

MSP SSP   WaitIters Mispredicts OverFlows Reconnect
--- --- ----------- ----------- --------- ---------
  0   0     3316440           0         1        24
  0   1   205831159       10207         0        22
  0   2   206147477       10207         0        23
  0   3   205925417       10207         0        23
```

The column headers have the following meanings:

MSP
: Number of the multistreaming processor. Since only one MSP is used, its number is 0.

SSP
: Number of each SSP. SSP 0 is the master processor.

UserSecs
: Wall-clock time spent outside of multistreamed code for each SSP. The master processor spent most of its time outside multistreamed code.

`MsLibSecs`	Wall-clock time spent inside the multistreaming library. When an SSP is inside the library, it is either waiting for a multistreamed section of code to be entered or resumed (SSPs are parked while waiting at a barrier) or an SSP is triggering the beginning or resumption of a multistreamed section of code.
`#MsEnts`	The number of times multistreamed procedures were entered. Entire procedures are multistreamed; within these procedures, SSPs 1, 2, and 3 skip past code that must be executed by one SSP.
`#Parks`	The number of times the MSP was parked so that SSP 0 could execute a single-streamed piece of code. Parking and unparking is the way SSPs 1, 2, and 3 skip around single-streamed code. The value is for the entire program.
`#Barrs`	The number of times the MSP encountered a streaming barrier in the program. Barriers are often needed for data synchronization or cache invalidation.
`#CInvs`	The number of cache invalidations. This must be less than or equal to the number of parks plus the number of barriers. Cache is invalidated for safety reasons.
`WaitIters`	SSPs execute *idle wait* loops while they are inside of the multistreaming library and waiting for other SSPs to synchronize with them. `WaitIters` is the total number of iterations each SSP spent in these loops. The numbers should correlate roughly with `MsLibSecs`.
`Mispredicts`	The streaming library attempts to predict where SSPs 1, 2, and 3 will be needed next after being parked, and it advances execution speculatively toward that point. The library safely prefetches data and instructions into cache and may execute instructions leading up to the unpark location. `Mispredicts` is the number of times this prediction was incorrect. It should be much less than the number of parks for best performance.

`Overflows`	The number of times the streaming library detected a common stack overflow. This should be nonzero only for SSP 0.
`Reconnect`	The approximate number of times each SSP was reconnected by the operating system. The values for the SSPs making up an MSP should be small and roughly equal, indicating long connect times and gang scheduling.

The statistics are gathered whether or not you have set the `MSP_STATS` environment variable. There is little overhead associated with gathering the statistics.

Optimizing Using Vectorization [4]

Vectorized constructs perform up to a factor of 20 times faster compared with nonvectorized constructs. Vectorization on Cray SV1 series systems does not always perform in the same way as on earlier Cray systems, however.

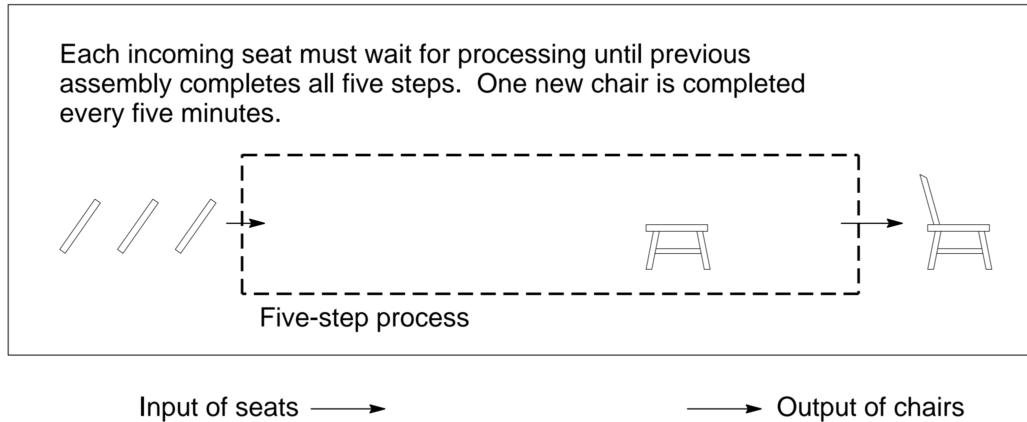
To make the best use of vectorization, you must first understand it. Section 4.1, page 39 describes how vectorization provides you with performance gains. If you already understand vectorization, proceed to Section 4.2, page 41.

Note: The timings in this chapter were from an early Cray SV1 system, not from a current Cray SV1ex system. They are presented to indicate the kind of improvement you can expect when you vectorize your code.

4.1 What Is Vectorization?

Vectorization is similar to an assembly line in manufacturing. To illustrate, Figure 7 compares two ways to carry out a five-step process for making chairs, in which each step takes 1 minute:

- In a scalar process, assembly must go through all five steps before the next assembly can begin the first step. One chair is produced every 5 minutes.
- In a vector process, each step in the sequence is performed as soon as the previous step is complete, like it is an assembly line. In this way, the first chair is completed after 5 minutes, and one new chair is completed each minute after that.



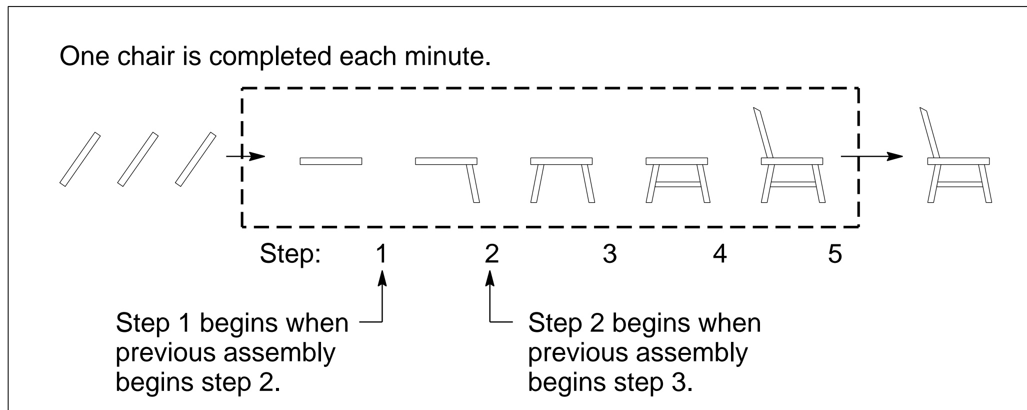


Figure 7. Scalar versus vector, illustrated

Like manufacturing a chair, processing an array in a loop is a multistep process that can be performed one step at a time, or by a process resembling an assembly line.

To process an array in either vector or scalar mode, a loop must:

- Load elements from memory to a register

- Process the elements, placing the results in another register
- Store the results back to memory

Conventional (scalar) code is a one-at-a-time process: each element must finish the final step of processing before the next element can begin the first step; that is, each loop iteration begins when the previous iteration ends. But with vector code, as in an assembly line, each element begins the first step when the previous element finishes the first step, rather than the final step.

4.2 Loopmark Listings

When you invoke the `-rm` option on the `ftn` command line, the Cray Fortran compiler will generate a loopmark listing. This loopmark listing displays at a high level what optimizations were performed by the compiler and tells you which loops were vectorized, parallelized, unrolled, interchanged, and so on.

The following example of a loopmark listing shows a loop that has been vectorized (V) and unrolled (r):

```
 1.     subroutine sub(nx,ny,nz,a,b,c)
 2.     real a(nx,ny,nz) ,b(nx,ny,nz) ,c(nx,ny,nz)
 3.
 4. 1------< do k = 2,nz
 5. 1 r----<  do j = 2,ny
 6. 1 r V--<   do i = 2,nx
 7. 1 r V     c(i,j,k) = 2.5*(a(i,j,k)-a(i,j-1,k))*(b(i,j,k)-b(i,j-1,k))
 8. 1 r V--<   end do
 9. 1 r----<  end do
10. 1------< end do
11.
12.     end

f90-6005 f90: SCALAR File = t.f, Line = 5
  A loop starting at line 5 was unrolled 2 times.

f90-6204 f90: VECTOR File = t.f, Line = 6
  A loop starting at line 6 was vectorized
```

In this example the loopmark indicates that the DO loop starting on line 6 has been vectorized and the DO loop starting on line 5 has been unrolled. The messages following the subroutine provide details concerning these optimizations. For example, the first message indicates that the loop starting at line 5 was unrolled twice.

To find out why a loop did not vectorize, use the `-Onegmsgs` option. In the above example, this option would generate the following additional messages:

```
f90-6294 f90: VECTOR File = t.f, Line = 4
  A loop starting at line 4 was not vectorized because
  a better candidate was found at line 6.

f90-6294 f90: VECTOR File = t.f, Line = 5
  A loop starting at line 5 was not vectorized
  because a better candidate was found at line 6.
```

Each loop is marked with one or more of the following codes, indicating the type of optimization performed:

Table 3. Loopmark Listing Keys

Primary Loop Type		Modifier	
A	pattern matched	b	blocked
C	collapsed	f	fused
D	deleted	i	interchanged
E	cloned	p	conditional, partial and/or computed
I	inlined	r	unrolled
M	multistreamed	s	shortloop
P	parallel/tasked	t	array syntax temp used
V	vectorized	w	unwound
W	unwound		

If no optimization is done to a loop, it is marked with a number indicating the nesting level, with the number `1` being the outermost loop.

If a loop is optimized, it is marked with the primary loop type and, optionally, one or more secondary modifiers, which indicate the type of optimization performed. For example, a loop may be marked with `V` to indicate that it was vectorized, with `Vr` to indicate that it was both vectorized (primary) and unrolled (modifier), or `r` to indicate that it was unrolled only. Modifiers are generally subordinate to primary loop types, but in some cases may be used alone. In other cases a loop may be subject to more than one primary optimization; for

example, a loop may be marked with `VM` to indicate that it was both vectorized and multistreamed.

The following sections discuss the different kinds of optimizations and the associated loopmark listing indicators.

4.3 Vectorization

The most common type of optimization performed by the compiler is vectorization. The following subsections discuss the different kinds of vectorization and their associated loopmark listing indicators.

4.3.1 Fully Vectorized Loops

The compiler may generate fully vectorized loops. A loop that is fully vectorized is indicated in loopmark listings with `V`, unless the modifier `p` is appended. The `p` indicates that less-than-full vectorization has been done, so less performance improvement has been achieved. In this case, examine the loop code. Partially vectorized loops often can be rewritten to permit full vectorization.

4.3.2 Partially Vectorized Loops

The compiler may generate partially vectorized loops. A loop is partially vectorized if it is split into multiple loops, at least one of which is scalar. This optimization is most effective on loops that contain a higher than usual percentage of vector work.

The following list contains portions of compiler messages that indicate partial vectorization. To obtain more detail on each message produced by the compilers, use the `explain`(1) command.

- `Partially vectorized`
- `Partially vectorized with a single vector iteration`
- `Partially vectorized with a vector length of` *N*
- `Partially vectorized with a computed safe vector length`
- `Partially and conditionally vectorized`

Partially vectorized loops are indicated with `Vp` in loopmark listings.

4.3.3 Conditionally Vectorized Loop

The compiler may generate a conditionally vectorized loop, which results in two loops; a scalar and a vector version. A runtime conditional expression chooses the scalar loop when needed to avoid a recurrence; otherwise, the vector loop is chosen. Here are two examples of loops that would be conditionally vectorized:

Fortran example:

```
do i=1,n
   if (s1.EQ.0) then
      a(i) = 0.0
   else
      a(i) = a(i-1)
   end if
end do
```

C++ example:

```
for (i=0; i<n; i++
if (a[i]!=0) {
a[i]=0.0
}else{
a[i]=a[i-1]
}
```

Conditionally vectorized loops are indicated with `Vp` in loopmark listings.

If you know that there will never be a recurrence in that loop, you can insert a `!dir$ ivdep` directive (Fortran) or a `#pragma _CRI ivdep` (C/C++) to avoid the extra logic and get a fully vectorized loop. Likewise, if you know that there will always be a recurrence, you can use the `!dir$ nextscalar` Fortran directive to prevent the conditional expression to get a faster scalar loop.

4.3.4 Reduction Loop

The compiler may generate a particular form of vectorization called a reduction loop. A reduction loop (also known as a summation loop or vector collapse) sums the elements of one array dimension. If the size of that dimension is considerably larger than the hardware vector register size, most of the summation can occur with partial sums in vector registers. Some scalar code is required to sum up the elements of the vector. The following samples illustrate loops that would be optimized as reduction loops:

Fortran example:

```
do i=1,n
   sum=sum+a(i)
end do
```

C++ example:

```
for (i=0; i<n; i++){
sum = sum+a[i];
}
```

Reduction loops are indicated with `V` in loopmark listings.

4.3.5 Shortloop

The compiler may generate a *shortloop*. A shortloop is a loop that is vectorized but also determined by the compiler to have fewer than 65 trips. In this case the compiler deletes the top test for number of trips, and deletes the loop to the top of the loop. A shortloop is more efficient than a conventional loop.

The compiler determines that a loop is a shortloop when:

- The `!dir$ shortloop` directive is used when compiling
- The constant trip count is less than 65
- Analysis of the array bounds shows that the array extent is declared to be less than 64

Shortloops are indicated with `Vs` in loopmark listings.

4.3.6 Vector Update Loop

The compiler may generate vector update loops. A vector update loop performs arithmetic on existing elements of an array and stores the results back into the same array. This type of loop requires both a gather operation from and a scatter operation to the same memory location and is not as fast as a fully vectorized loop. The following samples illustrate code that would be optimized as vector update loops:

Fortran example:

```
do i=1,n
   a(ib(i)) = a(ib(i))+b(i)
end do
```

C++ example:

```
for (i=0; i<n; i++) {
a[ib[i]]-a[ib[i]]+b[i];
}
```

Update loops are indicated with `Vp` in loopmark listings.

To improve its performance, insert an `IVDEP` directive, but only if you know there are no subscript collisions.

4.3.7 Computed-Safe Vector Length Loop

The compiler might generate a vector loop with a computed-safe vector length (VL) to avoid a recurrence. The safe VL is generated at runtime. If the computed-safe VL creates a fully vectorized loop (that is, it is greater than or equal to the machine-maximum VL), it is still moderately slower than its unconditional fully vectorized counterpart because of the overhead involved in the safe VL computation.

Examples of defining computed-safe VL loops follow:

Fortran example:

```
do i=1,n
   a(i) = a(i-m)
end do
```

C++ example:

```
for (i=0; i<n; i++) {
a[i]=a[i-m];
}
```

Loops vectorized with computed-safe VL are indicated with `Vp` in loopmark listings.

If you know that there will be no dependencies caused by array index overlap between loop iterations, you can avoid this overhead by using the `ivdep` directive to force this into a fully vectorized loop.

4.3.8 Vectorization Inhibitors

For loops with a very large (hundreds) number of lines, use the `-O aggress` optimization option on the `ftn`(1) compiler line. This option will allow for larger internal tables for compiler analysis.

Some loops will not vectorize with the default value of `vector2`. In these cases, using the `-O vector3` option on the `ftn`(1) compiler line may improve optimization.

Vectorization inhibitors within DO loops include:

- `CALL` statements not inlined
- I/O statements
- Backward branches
- Statement numbers with references from outside the loop
- References to character variables
- External functions that do not vectorize
- `RETURN`, `STOP`, or `PAUSE` statements
- Dependencies (see the following section)

You can address many of these inhibitors by slightly modifying the source code.

4.3.9 Vectorization and Dependencies

The compiler analyzes loops for dependencies. If a forward dependency is found (data is read, then written), the loop can be vectorized. For example:

```
do i = 1,n
   a(i) = a(i+1) * b(i)
end do
```

If you know that a loop is free of dependencies but the compiler cannot determine this, use the `!dir$ ivdep` directive (Fortran) or a `#pragma _CRI ivdep` (C/C++) to tell the compiler that the loop contains no vector dependencies. A common example is illustrated in the update loop shown below.

The compiler must assume that there will be collisions of indices and thus the loop is vectorized with considerable overhead, as shown by the loopmark listing:

```
 6.  Vp----<   DO i = 1,n
 7.  VP r-<>    e(ix1(i)) = e(ix1(i)) - a(i)
 8.  VP---->   END DO
 9.
10.       end

f90-6371 f90: VECTOR File = gs-2.f, Line = 6
  A vectorized loop contains potential conflicts due to indirect
  addressing at line 7, causing less efficient code to be generated.

f90-6204 f90: VECTOR File = gs-2.f, Line = 6
  A loop starting at line 6 was vectorized.
```

When you know that there will be no collisions, you can insert an `ivdep` directive to yield considerable performance improvement; for example:

```
    6.       !dir$ ivdep
    7.   V--<      DO i = 1, n
    8.   V           e(ix1(i)) = e(ix1(i)) - a(i)
    9.   V-->      END DO
   10.
   11.             end

f90-6203 f90: VECTOR File = gs-2.f, Line = 7
  A loop starting at line 7 was vectorized because an IVDEP
  or CONCURRENT compiler directive was specified.
```

4.4 Other Loop Optimizations

In addition to vectorization there are a number of other loop optimizations that may be performed by the compiler. These other optimizations include:

- Outer-loop vectorization
- Unrolling
- Unwinding
- Interchange
- Collapse

- Cloning
- Fusion
- Pattern matching
- Autotasking
- Streaming
- Blocking

4.4.1 Outer-loop Vectorization

Historically, vectorization has been done on inner loops only. Cray compilers have the ability to vectorize either inner or non-inner (outer) loops.

As an example, the outer loop below (`do i`) is vectorized. Note that array `a(i)` is invariant with respect to the inner loop (`do j`). The load of `a(i)` is done before the inner loop and the store is done following it. This reduces the memory traffic within the `do j` loop by two memory operations and increase performance significantly.

```
    4.  V----<      do i = 1,m
    5.  V r--<           do j = 1,n
    6.  V r                    a(i) = a(i) + b(i,j)*c(j)
    7.  V r-->           end do
    8.  V---->      end do
    9.
   10.               end

ftn-6204 f90: VECTOR File = outer.f, Line = 4
  A loop starting at line 4 was vectorized.

ftn-6005 f90: SCALAR File = outer.f, Line = 5
  A loop starting at line 5 was unrolled 2 times.
```

4.4.2 Loop Unrolling

Loop unrolling is done for the following reasons:

- More efficient use of vector operations
- Vector register reuse

- Cache reuse if all reused data doesn't fit in vector registers
- Reduced number of executed branch instructions and other loop overhead

A non-vector loop is chosen to be unrolled when a pattern of *nearest neighbor* (for example, `a(i,j) = a(i,j+1)`) is seen for that loop. Unrolling a non-vector loop is referred to as *unroll-and-jam*. The nearest neighbor pattern can be either write-read or read-read. Unroll-and-jam is done primarily to improve register and cache re-use. Reusing vector registers or cache reduces bandwidth usage, thereby improving performance.

For example, consider the following unroll-and-jam optimization:

```
 5.  r-----<       do j = 1,n-1
 6.  r V---<         do i = 1,n      ! vectorize
 7.  r V              a(i,j) = b(i,j) + b(i,j+1)
 8.  r V--->         enddo
 9.  r----->       enddo

f90-6005 f90: SCALAR File = nn-3.f, Line = 5
  A loop starting at line 5 was unrolled 2 times.

f90-6204 f90: VECTOR File = nn-3.f, Line = 6
  A loop starting at line 6 was vectorized.
```

The loopmark character `r` indicates unrolling. The loop is converted to:

```
do j = 1,n-1,2    ! unrolled for reuse of b(i,j+1)
  do i = 1,n      ! vectorized
    a(i,j)   = b(i,j)   + b(i,j+1)
    a(i,j+1) = b(i,j+1) + b(i,j+2)
  end do
end do
```

Data `b(i,j+1)` is loaded once but used twice in the unrolled version.

Unrolling of the vector loop is done for very small loops, to promote the use of more vector registers and operations in parallel.

Loop unwinding is a special case of unrolling, where the loop is entirely unrolled. For example:

```
    4.  V----<       do j = 1,n
    5.  V W--<         do i = 1,3
    6.  V W              a(i,j) = b(i,j) + c(i,j) * s1
    7.  V W-->         end do
    8.  V---->       end do
    9.
   10.               end

f90-6204 f90: VECTOR File = unwind-1.f, Line = 4
  A loop starting at line 4 was vectorized.

f90-6008 f90: SCALAR File = unwind-1.f, Line = 5
  A loop starting at line 5 was unwound.
```

The loopmark character w indicates unwinding.

4.4.3 Loop Interchange

Loop interchange is done to promote vector register re-use and quicker cache re-use. As a result of interchange, there is additional outer-loop vectorization, resulting in invariant vector registers and operations. These vectors can be hoisted and reused within the inner loop, which is very effective in reducing bandwidth usage.

The following example illustrates interchange in matrix-vector multiplication:

```
   25.  ir----<       do j = 1,m
   26.  ir V--<         do i = 1,n
   27.  ir V              a(i) = a(i) + b(i,j) * c(j)
   28.  ir V-->         end do
   29.  ir---->       end do

f90-6007 f90: SCALAR File = dgemv-1.f, Line = 25
  A loop starting at line 25 was interchanged with
  the loop starting at line 26.

f90-6005 f90: SCALAR File = dgemv-1.f, Line = 25
  A loop starting at line 25 was unrolled 2 times.

f90-6204 f90: VECTOR File = dgemv-1.f, Line = 26
  A loop starting at line 26 was vectorized.
```

The loopmark character `i` indicates that the loop was interchanged. The inner loop is converted to a dot product:

```
do i = i,n        ! vectorized
  do j = i,n
      a(i) = a(i) + b(i,j) * c(j)
  end do
 end do
```

The vector load of `a(i)` can be loaded ahead of the j-loop and the vector store of `a(i)` can be pushed below. Within the inner loop, instead of two vector-loads and one vector-store, there will be one vector-load and no vector-stores. Hoisting of vector registers is limited on the Cray SV1 series of systems because the systems have only 8 vector registers.

4.4.4 Loop Collapse

Loop collapse combines two or more loops into a single loop. This produces:

- Longer vectors
- Less loop overhead
- Better tasking load balance

For example, consider this loop:

```
    1.                subroutine sub(n)
    2.                common/blk/s1,a(100,100),b(100,100),c(100,100)
    3.
    4.  C-----<       do j = 1,100
    5.  C Vr--<         do i = 1,100
    6.  C Vr              a(i,j) = b(i,j) + c(i,j) * s1
    7.  C Vr-->         end do
    8.  C----->       end do
    9.
   10.                end

f90-6003 f90: SCALAR File = unwind-1.f, Line = 4
  A loop starting at line 4 was collapsed into
  the loop starting at line 5.

f90-8135 f90: SCALAR File = unwind-1.f, Line = 5
  Loop starting at line 5 was unrolled 2 times.

f90-6204 f90: VECTOR File = unwind-1.f, Line = 5
  A loop starting at line 5 was vectorized.
```

The loopmark `C` indicates loop collapse. Essentially, the loop is converted to:

```
do i = 1,100 * 100
  a(i,1) = b(i,1) + c(i,1) * s1
end do
```

4.4.5 Loop Fusion

Loop fusion combines two or more loops into a single loop. The goals are reduced loop overhead and register reuse. Loop fusion is very useful for optimizing Fortran array syntax, where each statement is an implied loop:

```
    1.                 subroutine sub
    2.                 common/blk/a(100),b(100)
    3.
    4.  V---<>         a=0.
    5.  f---<>         b=0.
    6.
    7.                 end

f90-6204 f90: VECTOR File = fusion.f, Line = 4
  A loop starting at line 4 was vectorized.

f90-6004 f90: SCALAR File = fusion.f, Line = 5
  A loop starting at line 5 was fused with the
  loop starting at line 4.
```

Loop fusion is also useful as illustrated in the case below, where array temp can be optimized away by using a register as the temporary instead of memory:

```
    5.  Vr--<        do i = 1,n
    6.  Vr             temp(i) = b(i) * c(i)
    7.  Vr-->        end do
    8.
    9.  f---<        do i = 1,n
   10.  f              a(i) = temp(i) + d(i)
   11.  f--->        end do

f90-8135 f90: SCALAR File = fusion.f, Line = 5
  Loop starting at line 5 was unrolled 2 times.

f90-6204 f90: VECTOR File = fusion.f, Line = 5
  A loop starting at line 5 was vectorized.

f90-6004 f90: SCALAR File = fusion.f, Line = 9
  A loop starting at line 9 was fused with the
  loop starting at line 5.
```

The loopmark character `f` indicates loop fusion.

4.4.6 Pattern Matching

Common patterns of blas-2, blas-3, and others are recognized and replaced with calls to libsci library routines. Among the recognized patterns are matrix-matrix multiply, matrix-vector multiply, sgerx, and some forms of isamin/isamax. The following example shows a matrix-matrix multiply that has been pattern matched:

```
   23.  D--------<       do k = 1,n3       !
   24.  D D------<         do j = 1,n2     !       M
   25.  D D D----<           do i = 1,n1   !       V
   26.  D D D A-<>             x(i,j) = x(i,j) + y(i,k) * z(k,j)
   27.  D D D---->           end do
   28.  D D------>         end do
   29.  D-------->       end do
   30.
   31.                   ir2=irtc()
   32.                   ir=ir2-ir1
   33.                   end

f90-6002 f90: SCALAR File = dgemm-1.f, Line = 23
  A loop starting at line 23 was eliminated by optimization.

f90-6002 f90: SCALAR File = dgemm-1.f, Line = 24
  A loop starting at line 24 was eliminated by optimization.

f90-6002 f90: SCALAR File = dgemm-1.f, Line = 25
  A loop starting at line 25 was eliminated by optimization.

f90-6202 f90: VECTOR File = dgemm-1.f, Line = 26
  A loop starting at line 26 was replaced by a library call.
```

The loopmark character `A` indicates that pattern-matching has occurred. The loopmark character `D` indicates that the loops have been eliminated.

4.4.7 Autotasking

When the options `-O task2` or `-O task3` are in effect, the loopmark character of `P` is used to indicate loops that are tasked:

```
   68.  P-----<       do j = 1,m
   69.  P V---<         do i = 1,n
   70.  P V               c(i,j) = a(i,j) + d(i,j)
   71.  P V               a(i,j) = b(i,j) * 10.0
   72.  P V--->         end do
   73.  P----->       end do

f90-6403 f90: TASKING File = 2-level.f, Line = 68
  A loop starting at line 68 was tasked.

f90-6204 f90: VECTOR File = 2-level.f, Line = 69
  A loop starting at line 69 was vectorized.
```

4.4.8 Streaming

When the options `-O stream2` or `-O stream3` are in effect, the loopmark character of `M` is used to indicate loops that are streamed:

```
   68.  M-----<       do j = 1,m
   69.  M V---<         do i = 1,n
   70.  M V               c(i,j) = a(i,j) + d(i,j)
   71.  M V               a(i,j) = b(i,j) * 10.0
   72.  M V--->         end do
   73.  M----->       end do
   74.                ir2=irtc()
   75.                ir=ir2-ir1
   76.                end

f90-6601 f90: STREAM File = 2-level.f, Line = 68
  A loop starting at line 68 was multistreamed.

f90-6204 f90: VECTOR File = 2-level.f, Line = 69
  A loop starting at line 69 was vectorized.
```

4.4.9 Loop Blocking

Cray compilers do not perform automatic loop blocking at present. However, the blockable directive is available to do manual blocking:

```
    4.  1--------<       do k = 1,n
    5.  1          !dir$   blockable (j,i)
    6.  1          !dir$   blocking size (20)
    7.  1 b------<         do j = 1,m
    8.  1 b        !dir$     blocking size (20)
    9.  1 b Vb---<           do i = 1,mm
   10.  1 b Vb                z(i,k) = z(i,k) + x(i,j) * y(j,k)
   11.  1 b Vb--->           end do
   12.  1 b------>         end do
   13.  1-------->       end do
   14.
   15.                   end

f90-6051 f90: SCALAR File = blockable.f, Line = 7
  A loop starting at line 7 was blocked according
  to user directive with block size 20.

f90-6051 f90: SCALAR File = blockable.f, Line = 9
  A loop starting at line 9 was blocked according
  to user directive with block size 20.

f90-6205 f90: VECTOR File = blockable.f, Line = 9
  A loop starting at line 9 was vectorized with a
  single vector iteration.
```

The loopmark character `b` indicates loop blocking. See the *Cray Fortran Compiler Commands and Directives* manual for more information.

Optimizing Memory Use [5]

This chapter gives an overview of memory and describes optimizing cache use (Section 5.2, page 62)and managing memory (Section 5.3, page 67).

5.1 Overview of Memory

This section describes central memory and cache.

5.1.1 Central Memory

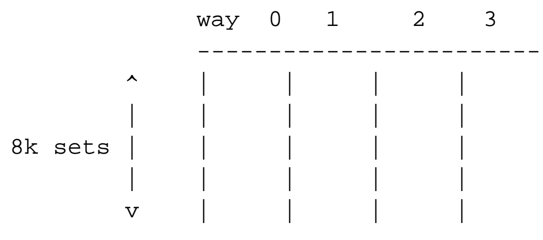
Table 4, page 59 gives typical latencies for the Cray SV1 series systems. The times are given in 300-MHz clock periods for Cray SV1 systems and 500-MHz clock periods for Cray SV1ex systems.

Table 4. Typical Latencies

Operation	Cray SV1 Systems	Cray SV1ex Systems
V register to memory	109	170
V register to cache	25	20
S register to cache	22	17
Floating-point add	8	8
Floating-point multiply	9	9
Floating-point reciprocal	16	16
Jump	6	6

5.1.2 Cache

The Cray SV1 series system cache size is 32 KW per single-streaming processor (SSP) and 4-way set associative. A referenced word, based on its address, is mapped to one of 8K *sets* and allocated to one of four *ways*.



The data resulting from vector, scalar, and instruction buffer memory references is cached. The cache line size is 8 bytes, or 1 word long. Cache is located between the processor registers and the interface to memory.

Cray SV1 cache differs from those on mainstream microprocessors in several important ways.

cache line width
: The cache line width for scalar and vector references is a single word (8 bytes). Unlike most microprocessors, contiguous memory references are not required in order to achieve full cache (memory) bandwidth.

bandwidth
: Data cache has very high memory bandwidth (up to 4 words per clock period).

size
: The Cray SV1 cache is 256 KB, which is small by modern standards. You may need smaller blocking factors than on many microprocessor-based systems.

write-through
: This means that any memory stores go all the way to memory. Because system memory bandwidth is often a limiting resource, it pays to minimize unnecessary data stores.

coherence
: Due to the need for binary compatibility, Cray SV1 system cache does not maintain coherency with the other processors. This means that if another processor performs a store to memory, the new data value is not in cache. When the

system synchronizes multiple processors, cache is invalidated (essentially, cleared). The implication is that better parallel performance is achieved with larger granularity.

A memory address is mapped in a manner that results in the data represented by that address being placed in one particular cache set. That is, in an *N*-way set-associative cache, an expression such as the following indicates to which set in cache the address maps:

`modulo (`*memory_address*`, `*cache_size*`/`*N*`)`

Since memory is much larger than cache, each set has many memory addresses that map into it. The four ways allow up to four memory addresses to map into the same cache set. If it becomes necessary to map a new address into a fully allocated cache set, the way for the least recently used address will be used and its data will be overwritten.

> **Note:** Direct-mapped cache and fully associative cache can be viewed as special end cases of the general *N*-way set-associative cache. For direct-mapped cache, *N* equals 1, and for fully-associative cache, *N* equals the number of cache lines.

The write allocate attribute of cache requires that the address and data for a processor write request to memory be mapped and placed into cache, if the request generates a cache miss. Because the Cray SV1 system has a write-through cache, any value that is updated in cache will always be written through to memory at the same time. This is in contrast to a write back cache, in which a value is updated in memory when there is no longer room for it in cache (that is, the value will be written back during a read to another value, which makes reads more expensive).

Write through can mean that more writes to memory are made than are strictly necessary when data is updated several times while remaining cache-resident. The cache contains buffers capable of holding 384 outstanding references. This is sufficient for the cache to handle 6 vector references of stride 1 (384 references = 6 vectors * 64 references per vector).

The 1-word cache line size is advantageous for non-unit vector strides, since it does not cause the overhead of unnecessary data traffic when referencing memory using larger cache line sizes.

The disadvantage is that a single scalar reference will not bring in surrounding data, thus potentially inhibiting a scalar code from taking advantage of spatial locality. For this reason, scalar references have a prefetch feature, whereby a scalar reference causes 8 words to be brought into cache. These 8 words are

determined by addresses that match the reference, except for the lower 3 bits. The effect for scalar loads is similar to having a cache line size of 8 words.

The Cray SV1 system relies on software that maintains cache coherency between processes that share memory. Cache is invalidated as part of the test-and-set instruction. The test-and-set instruction has been used for processor synchronization in previous generation Cray vector systems. Adding the cache invalidation feature to this instruction allows old Cray binaries to run in parallel on the Cray SV1 system with the data cache enabled.

On Cray SV1ex systems, cache is not always cleared at a test-and-set instruction. When it is determined to be safe, the compiler uses a new instruction that disables cache invalidation. Doing so allows access to the data that is already in cache and thereby improves performance.

Data and instructions are cached by default. To turn caching off, use a command such as the following:

```
% /etc/cpu -m ecfoff a.out
```

For more information, see the cpu(8) man page.

5.2 Optimizing Cache Use

The following cache optimization techniques are described in this section:

- Using vector cache effectively (Section 5.2.1, page 62)
- Minimizing stores (Section 5.2.2, page 63)
- Porting issues (Section 5.2.3, page 66)

5.2.1 Using Vector Cache Effectively

How can you tell whether a program is using data cache effectively? The hardware performance monitor (hpm(1)) gives you the number of reads that are hitting in cache. (The following output is reformatted for readability):

```
% hpm ./a.out Group 0:
  CPU seconds             : 0.05702
  CP executing            : 17107068
  ....
  Cache hits/sec          : 70.77M
  Cache hits              : 4035458
  CPU mem. references/sec : 233.45M
```

```
  CPU references          : 13312268

  Floating ops/CPU second : 157.83M
```

You can see how cache usage is affecting the performance of your code by running with the cache turned off, using the `-m ecdoff` option of the `/etc/cpu`(1) command. This is the same program as the previous example:

```
% /etc/cpu -m ecdoff hpm ./a.out Group 0:
  CPU seconds             : 0.10927
  CP executing            : 32780370
  ...
  Cache hits/sec          : 0.01M
  Cache hits              : 608
  CPU mem. references/sec : 121.83M
  CPU references          : 13312268
```

Turning off cache approximately doubled the performance of this program.

Cray SV1 series systems have the same memory hardware as the Cray J90 system with the same memory speed, despite having a processor with six times the peak performance. This means that it is possible for memory to be oversubscribed by the processors; that is, the performance of a processor running a memory-intensive application will be adversely affected if other processors are also using memory intensively because of contention for the limited number of physical ports to memory. Because of this fact, and because of cache, users should expect much more variability in programming timings than on earlier UNICOS systems.

The following are techniques for optimizing cache:

- Avoid memory accesses with strides that are a multiple of 8192, since four of these will fill up the cache slot.
- In general, avoid memory accesses whose strides are large powers of two. Only eight loads of stride 4096 and sixteen of stride 2048 will fit in cache.
- Strides of one are optimal.

5.2.2 Minimizing Stores

The following matrix-vector multiply kernel illustrates how to minimize stores. A matrix vector multiply of size N has $2*N^2$ floating-point operations and N^2

data. From an algorithm perspective, a minimum of one memory operation will be required for every two floating-point operations, giving a maximum computational intensity of two (two flops per memory operation).

To inhibit full compiler optimization, compile the loop as follows:

```
% ftn -O nopattern,nointerchange -rm mxv.f
```

```
                 subroutine mxv(a,lda,n,b,x)
                 real a(lda,n), b(lda), x(lda)
 1-----<         do j = 1, n
 1 Vr--<           do i = 1, n
 1 Vr              x(i) = x(i) + a(i,j)  *b(j)
 1 Vr-->           enddo
 1----->         enddo
                 return
                 end
```

From the listing file, the inner loop is vectorized (V) and unrolled (r). The unrolling here is on a vector chime basis (64 elements), not on a iteration-by-iteration basis. For example, if the compiler unrolls a vector loop by two, it means two vector chimes (or 128 elements) are processed before loop iterations are incremented.

HPM shows that this loop runs at 250 MFLOPS and is requesting operands at the rate of 374 MWps (3 words for every 2 FLOPS). Of this, 250 MWps is satisfied by main memory and the remaining 124 MWps is satisfied from data cache.

Since `x(i)` is updated for each pass of `j`, we can unroll the `j` loop into the inner loop and reduce the number of times `x(i)` is updated. For example, if we unroll by four times, we should reduce loads and stores to `x` by a factor of four:

```
                subroutine mxv1(a,lda,n,b,x)
                real a(lda,n), b(lda), x(lda)
 1----<         do j = 1, n, 4
 1 V--<         do i = 1, n
 1 V            x(i) = x(i) + a(i,j)  *b(j)
 1 V        1               + a(i,j+1)*b(j)
 1 V        1               + a(i,j+2)*b(j)
 1 V        1               + a(i,j+3)*b(j)
 1 V-->         enddo
 1---->         enddo
                return
                end
```

Performance has improved from 250 to 301 MFLOPS. In addition, the overall bandwidth consumed has dropped from 374 MWps to 226 MWps (38 MWps from cache and 188 MWps from main memory).

Another alternative is to switch the loop ordering and go to a dot-product formulation. This eliminates vector store traffic in the inner loop:

```
                subroutine mxv(a,lda,n,b,x)
                real a(lda,n), b(lda), x(lda)
1-----<         do i = 1, n
1        cdir$ prefer vector
1 Vr---<          do j = 1, n
1 Vr                x(i) = x(i) + a(i,j)  *b(j)
1 Vr--->          enddo
1----->         enddo
                return
                end
```

This formulation runs at 298 MFLOPS, consuming 306 MWps of total bandwidth (153 MWps from cache and 153 MWps from memory). In this case, loads to `b(j)` are cached and loads to `a(i,j)` are not. We have reached our algorithmic ideal ratio of two flops for every memory operation. Dot products, however, are not ideal on vector systems due to the final vector reduction operation, in which a vector register of operands is collapsed down to a single scalar value.

The ideal algorithm would hold 64 elements of `x` in a vector register until all updates are complete. The 64 elements of the completed vector `x` would be written to memory. This technique is called *outer-loop vectorization*. It can be achieved by writing the loop in a dot-product formulation and then inserting a `cdir$ prefer vector` directive before the outer loop:

```
                subroutine mxv(a,lda,n,b,x)
                real a(lda,n), b(lda), x(lda)
         cdir$ prefer vector
V-----<         do i = 1, n
V r---<           do j = 1, n
V r               x(i) = x(i) + a(i,j)  *b(j)
V r--->           enddo
V----->         enddo
                return
                end
```

This loop now runs at 395 MFLOPS, consuming only 201 MWps of total bandwidth (199 MWps from memory and 2 MWps from cache). Reuse has

moved from cache into a vector register. In many cases the compiler will choose this formulation automatically.

The Cray SV1 system memory is capable of delivering operands at the rate of 2.5 GBps (312 MWps). Since two flops are computed for every word of memory bandwidth, this algorithm has a peak potential rate of 624 MFLOPS (312 MWps * 2) on the Cray SV1 system. When coded in assembly language, vector loads to `a` can be carefully scheduled to achieve maximum bandwidth:

```
      subroutine mxv(a,lda,n,b,x)
      real a(lda,n), b(lda), x(lda)
      CALL SGEMV ('n', N, N, 1., A(1,1), LDA,
$                 B(1), 1, 1., X(1), 1)
      return
      end
```

The scientific libraries code runs this problem at 600 MFLOPS, consuming 305 MWps of memory bandwidth (3 MWps from cache and 302 MWps from memory).

5.2.3 Porting Issues

The Cray SV1 series system libraries for parallel processing have all been modified to invoke the test-and-set instruction when cache must be invalidated. Therefore, you should be able to port codes written for Autotasking, OpenMP, and MPI without modification.

SHMEM codes should also port without modification, with one exception. Autotasking and OpenMP use a test-and-set instruction that is issued at the beginning and end of parallel regions, at parallel loop iterations, and at critical regions (locks and guards).

For MPI programs, there is no shared data from the user's view, and the libraries take care of managing consistency within themselves. Although it is better to maximize the granularity of parallel tasks and minimize synchronization on any architecture, there is additional incentive for the Cray SV1 series system because it is best to avoid cache invalidations.

The SHMEM library routines `shmem_barrier`(3), `shmem_wait`(3), and `shmem_udcflush`(3) perform cache invalidations for the Cray SV1 series system. In order to avoid race conditions, a `shmem_barrier` or `shmem_wait` is usually issued before remotely updated data is used.

That will take care of cache coherency considerations at the same time. Codes that were originally written for the Cray T3D computer may need

to be modified if they make use of the `shmem_set_cache_inv`(3) and `shmem_clear_cache_inv`(3) routines. These routines invalidated cache on the Cray T3D system and are ignored on the Cray T3E computer, but they are not supported on the Cray SV1 system; you will get an unsatisfied external message when you try to load.

These codes will need to be reworked, probably by replacing each `shmem_set` or `clear_cache_inv` call with a `shmem_barrier` or `shmem_udcflush`.

5.3 Managing Memory

The Cray SV1 system contains fixed-size, real memory; it has no virtual memory capabilities. Large memory codes must fit within available user memory. User processes that dynamically expand and shrink during execution are managing memory, and they must compete with other processes for real memory space. Processes that spend excessive time managing memory are said to be *memory bound*.

Inefficient memory management within a process can increase its elapsed time, and this can affect other user processes in the system. If the process is large compared to the amount of available user memory, small performance problems can become worse. When a process attempts to dynamically change its size, it becomes a candidate for a swap out of memory onto an external device (such as a disk) before the operating system can find enough contiguous memory to swap it back in. Also, the code might make system memory calls without your knowledge. These situations can greatly increase elapsed time.

Optimizing memory-bound code reduces elapsed time and system CPU time, and it may have a side effect of reducing user CPU time.

> **Note:** Most of the memory management techniques described in this chapter reduce elapsed time at the cost of increased memory usage.

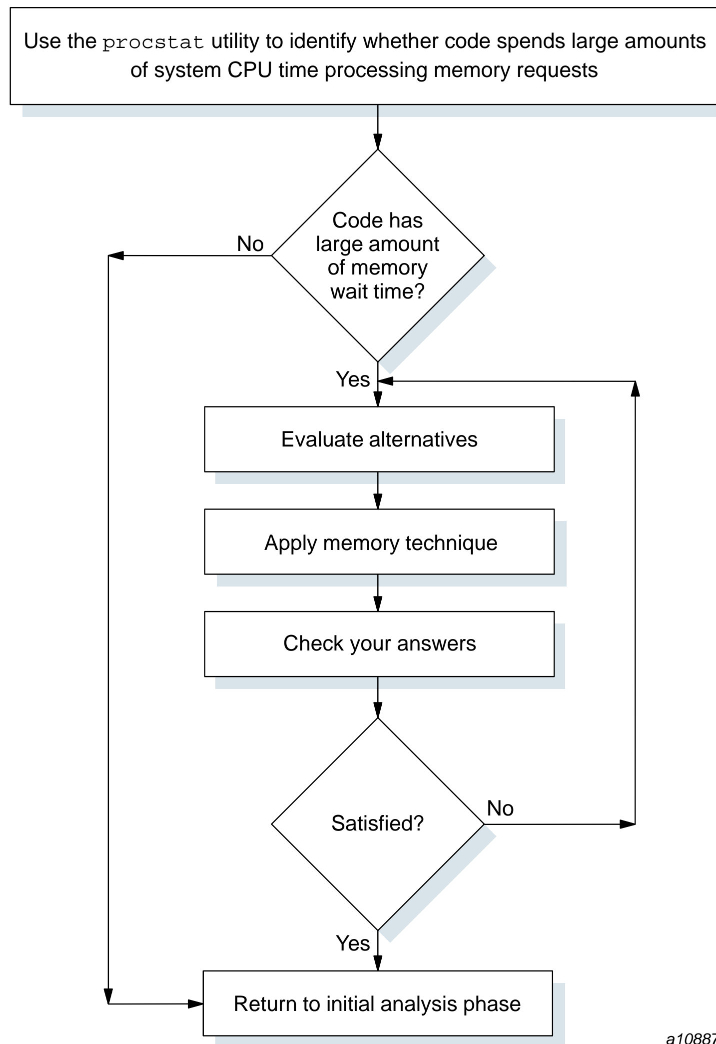


Figure 8. Optimizing memory-bound code

5.4 Understanding Memory Management

Two basic types of dynamic memory management are available on Cray SV1 series systems:

- Dynamic memory managed by the system heap routines

- Expandable dynamic common blocks

You cannot use both methods within the same code.

5.4.1 Dynamic Heap

When code is compiled and loaded, it is translated into Cray machine instructions (object code), logically linked together with library routines, and packaged in an executable file, named `a.out` by default. When you execute `a.out`, it is swapped into memory and becomes a *user process*. Most UNIX systems have at least the following three distinct areas defined in memory for each user process:

- User area
- Stack area
- Heap

The *user area* is where the object code (text area) and external and static variables (data area) reside. The *stack* area is commonly used to temporarily store context information for each routine that calls another subprogram. The *heap* is a dynamic area used for all other memory needs (except Fortran `COMMON`): dynamic variables, I/O buffers, flexible file input/output (FFIO) user cache, and so on.

The heap and the stack areas are allowed to grow dynamically, but the user area remains fixed. On a virtual memory system, both the heap and the stack area of a single process can grow independently with a virtual hole between them.

The UNICOS operating system implements dynamic allocation of heap and stack space differently from virtual systems. A UNICOS user process has a dynamically expandable heap, with stack space wholly contained within the heap. The stack space is managed without benefit of direct hardware support, and both heap and stack space must appear to grow and shrink independently.

Initial memory allocation for the heap and the stack space is established at the load step by the `ld`(1) utility (called by the `ftn`(1), `CC`(1), and `cc`(1) commands).

The heap is dynamic and can be increased or decreased from an executing code by using calls to the library. Routines request space from the heap directly through calls to the UNICOS system.

Dynamic memory management is inherently expensive to a user process because it requires service from the operating system through system calls. An expansion of the heap might require the process to be relocated in memory. If there is no remaining space large enough in memory, the UNICOS operating system will

swap the requesting process to a secondary device (such as a disk) until enough memory becomes available. This adds elapsed time and system CPU time.

Stack space must be allocated specifically in finite segments, or *stack frames*, within the heap. A stack frame is used as a place holder for procedure calls to save context, local variables, local arrays, and so on. Each procedure call pushes (allocates) a new frame on the stack, and pops (deallocates) that frame upon exit. The stack is a last-in, first-out (LIFO) model allocated inside the heap in finite-sized segments. Therefore, the stack frames compete with the rest of the dynamic memory requirements for space.

The following conditions can cause excessive system time associated with managing memory for your user process:

- Releasing heap blocks in a different order from that of their allocations can cause memory fragmentation for the process.
- Upon entry to a function, routine, or procedure in a user process, it might not be possible to allocate a stack frame within the boundaries of the current stack segment. This is a stack overflow condition (transparent to programmers) that causes attempts to allocate an additional stack segment within the heap.
- If a new stack segment cannot fit contiguously within the heap, total stack space becomes fragmented. This costs extra time for subroutine calls until that stack segment is no longer needed and returns to the heap (a stack underflow condition).
- *stack thrashing* is a rare condition in which a frequently called function allocates (stack overflow) and deallocates (stack underflow) both a stack segment and a heap block for every invocation.
- Fortran automatic arrays and C variable-length arrays are allocated with compiler-generated heap requests upon entry to a routine and deallocated upon exit from the routine. These arrays cause hidden memory management from system calls.
- Multitasked processes share a heap between multiple slave processes, each using its own stack space in the heap.

5.4.2 Dynamic Common Blocks

The UNICOS operating system provides a second method to manage memory for Fortran codes, the dynamic common block. To use this method, you must specify only one dynamic common block (which might be blank common) for the loader to place at the high end of the process memory space. For an example of

an `ftn` command line that establishes a dynamic common block, see Example 3 in Section 5.7.1, page 74.

This technique requires your heap to be a fixed size. Heap expansion is not allowed because the dynamic common area is stored directly after the heap. Therefore, the initial size of the heap must be large enough to handle all requests for heap space. Generally, an initial heap size between 5,000 and 10,000 words is adequate.

Fortran codes that use this method typically overindex an array within a dynamic common block, but the programmer must be careful to avoid an operand range error. You can expand and contract the dynamic common block by using the `SBREAK` Fortran library routine (see the `brk`(2) man page for more information). `SBREAK` expands the field length of the user process to provide more memory, and it also releases memory when it receives a negative argument.

All subroutines within the same code have access to its dynamic common block at any time during program execution. Its contents cannot be initialized at load or compile time.

5.5 Identifying Large Amounts of Memory Wait Time or System CPU Time

The `procstat`(1) utility can provide accurate memory information about your code. It gathers process-level memory statistics, such as elapsed time, number of calls to memory processor, number of memory declines, and total time to complete memory requests.

To create a report, execute the `procstat` utility with the name of the program to be analyzed listed as an argument.

For complete information on `procstat`, see the `procstat`(1) man page.

Procedure 4: Creating a Report

Use the following procedure to view a report of complete I/O information for your code:

1. Compile and run the code.
2. Run the `procstat` utility on the code.

The following examples show how to perform this procedure:

Fortran example:

```
ftn prog.f
procstat -r -m a.out
```

C++ example:

```
CC prog.C
procstat -r -m a.out
```

The `procstat` utility creates a viewable file with performance information.

5.6 Evaluating Dynamic Memory Alternatives and Applying a Technique

If your code is memory bound, it will probably exhibit one of the symptoms listed in the following sections. Each section lists a symptom followed by a recommended technique to reduce the elapsed time caused by memory requests. Evaluate the behavior of your code to see if it matches one or more of these symptoms, and select one of the corresponding techniques. After applying any optimization technique, check your answers and examine the new elapsed time for the code.

5.6.1 Large Number of System Calls

Check the number in the `Number of Calls to Memory Processor` row in the Procstat Process Report that you created. Does the code have a large number of system calls? If possible, reduce the number of system calls for memory within the source code.

5.6.2 Memory Expanded or Contracted in Small Increments

Check the number in the `Number of Calls to Memory Processor` row in the Procstat Process Report. Does the code still make many system requests to expand or contract memory? The requests may be too small.

If so, apply one of the following techniques:

- Reduce the number of memory requests and increase the size of the requests within the code.
- Ensure that the first system call requests sufficient memory for prolonged usage, and minimize system calls to shrink the size of the process.
- Initialize a larger heap, as described in Section 5.7, page 74.

5.6.3 Other Reasons for Excessive Memory Activity

If the Procstat Process Report still shows a large number of calls to the memory processor, the situation can be caused by any of the following situations:

- Alternate requests and releases of memory from the heap. Apply one of the following techniques:
  - Within your source code, attempt to reuse existing heap space instead of releasing it. Use the Fortran `ALLOCATE` and `DEALLOCATE` keywords or the C++ `new` and `delete` operators or the `malloc` or `free` library functions. These do not require the compiler to generate a system call.
  - In both Fortran and C++, avoid calling `sbreak` with a negative argument.
  - Initialize a larger heap, as described in Section 5.7, page 74.
- Frequent stack overflows and underflows (or *stack thrashing*). You can check this condition by using the Fortran `STOP` statement or the C++ `stkstat`(3) system call to produce a report of stack overflows. To avoid stack overflows, use the SEGLDR directive to increase the value in the `Initial stack size` to the maximum stack size as displayed by the STOP statement output. The `Initial stack size` is illustrated in Figure 9, page 77. To create the Load Map Program Statistics report, see Section 5.7.2, page 75.

  The following `STOP` statement output shows that the program experiences a large number of stack overflows:

  ```
  ftn where.f
  ./a.out < INPUT_FILE

  STOP   executed at line 261 in Fortran routine 'CLACIER'
  CP: 34.435s, Wallclock; 198.094s, 2.2% of 8-CPU Machine
  HWM mem: 236775, HWM stack: 10048, Stack overflows: 750000
  ```

  By indicating a stack size that matches the stack high water mark (shown by the `HWM stack` value), stack overflows are now lower, and CPU time improves from 34.4 seconds to 13.5 seconds:

  ```
  ftn -Wl"-S10048" where.f
  ./a.out < INPUT_FILE

  STOP   executed at line 261 in Fortran routine 'CLACIER'
  CP: 13.543s, Wallclock; 58.227s, 2.9% of 8-CPU Machine
  HWM mem: 209338, HWM stack: 10048, Stack overflows: 0
  ```

5.6.4 Temporary Memory Expansion of Significant Duration

When the process expands temporarily during execution, it runs the risk of being swapped to disk, costing excessive elapsed time. Use a SEGLDR directive (see Section 5.7, page 74) to initialize the process at its largest size to avoid a swap.

5.6.5 Heap Blocks Release Order

Examine the source code. Does the process release heap blocks in a different order than they were allocated? This can cause memory fragmentation for the process. Reorder the code to allocate and deallocate heap space in the opposite order (last allocated should be first deallocated).

5.7 Memory Initialization

When forced to obtain more memory within the code, it is more efficient to use a few large requests than to use many small requests. One way to do this is to directly modify source code to issue fewer, larger requests. Another way is to initialize the heap with a SEGLDR directive. The most effective way to minimize the elapsed time due to memory management is to start with enough memory in the first place.

5.7.1 Loader Directives

At the load step, SEGLDR, which is usually called by both the `ftn`(1) and `CC`(1) commands, establishes initial memory allocation for both the heap and the stack space. You can specify any of these parameters with a command-line option for `segldr`, `CC`, or `ftn`, as in the following examples.

Example 1:

The following SEGLDR command specifies an initial heap size of 150,000 words and a heap increment of 75,000 words for the object file, `file.o`:

```
segldr -H150000+75000 file.o
```

Example 2:

The following Cray C++ compiler command specifies an initial stack size of 100,000 words and stack increment of 50,000 words for the source file, `file.C`:

```
CC -dSTACK=100000+50000 file.C
```

Example 3:

The following Fortran compiler command names a dynamic common block (DYNAM), specifies an initial heap size of 10,000 words, and establishes a zero heap increment size for the source file, code.f. The heap increment is set to zero to force a fixed heap size required by using a dynamic common block:

```
ftn -Wl"-H10000+0;DYNAMIC=DYNAM" code.f
```

For more information on SEGLDR, see the *Segment Loader (SEGLDR) and ld Reference Manual*.

5.7.2 Optimal Heap Size

An optimal heap size for your process is a size that is large enough to prevent system calls for memory, but not so large that it contains unused memory.

Procedure 5: Determining Optimal Heap Size

To determine an optimal initial heap size for your SEGLDR directive, perform the following steps:

1. Generate a ja report or retrieve the ja report you used in your initial analysis in Procedure 2, page 20.
2. Create a Load Map Program Statistics report for the code by using SEGLDR directives.

   Example:

   The sample load map in Figure 9, page 77, was generated by using the following ftn command:

   ```
   ftn -Wl"-H10000+9000 -S6000+5000 -Mmap.fil,stat" code.f
   ```

   This command line specifies the following:

   - Load map statistics in a file called map.fil
   - Initial heap size of 10 KW
   - Heap increment of 9 KW
   - Initial stack size of 6 KW
   - Stack increment size of 5 KW
3. Look at the number in the Memory HiWater column of the ja report you created for the code (Procedure 2, page 20). This number is reported in blocks, and a block is equal to 512 (decimal) Cray memory words. Multiply

this figure by 512 to determine the maximum process memory size for the code in Cray words.

Example:

Assume the `Memory HiWater` figure from the `ja` report and the load map in Figure 9, page 77, are from the same code, `bigio`. This user process grew to 8,128 blocks. Perform the following arithmetic to obtain decimal words for `Memory HiWater`: 8,128 x 512 = 4,161,536 words.

4. Look at the number listed in the `Base address of managed memory/stack` row of the Load Map Program Statistics report (Figure 9, page 77). This number is an octal representation of the user area size measured in words. Convert this number to decimal to determine the user area size for the code in decimal format.

   Example:

   The `Base address of managed memory/stack` figure is 376,372 octal. This is equivalent to 130,298 decimal.

5. Subtract the user area size (total from the preceding step) from the maximum process memory size (total from Procedure 5, step 3, page 75). The result should be a good estimate for the minimum heap size required to avoid system calls for more memory.

   Example:

   4,161,536 - 130,298 = 4,034,238

   This represents the largest heap size for the process. Note that this might change for a different dataset, and it will expand with larger library I/O buffer and user cache sizes.

6. Use the result from the preceding step in a SEGLDR directive to set the initial heap size.

   Example:

   To avoid system requests for memory, relink the object file with the following `segldr` command:

```
segldr -H4035000+10000 bigio.o
```

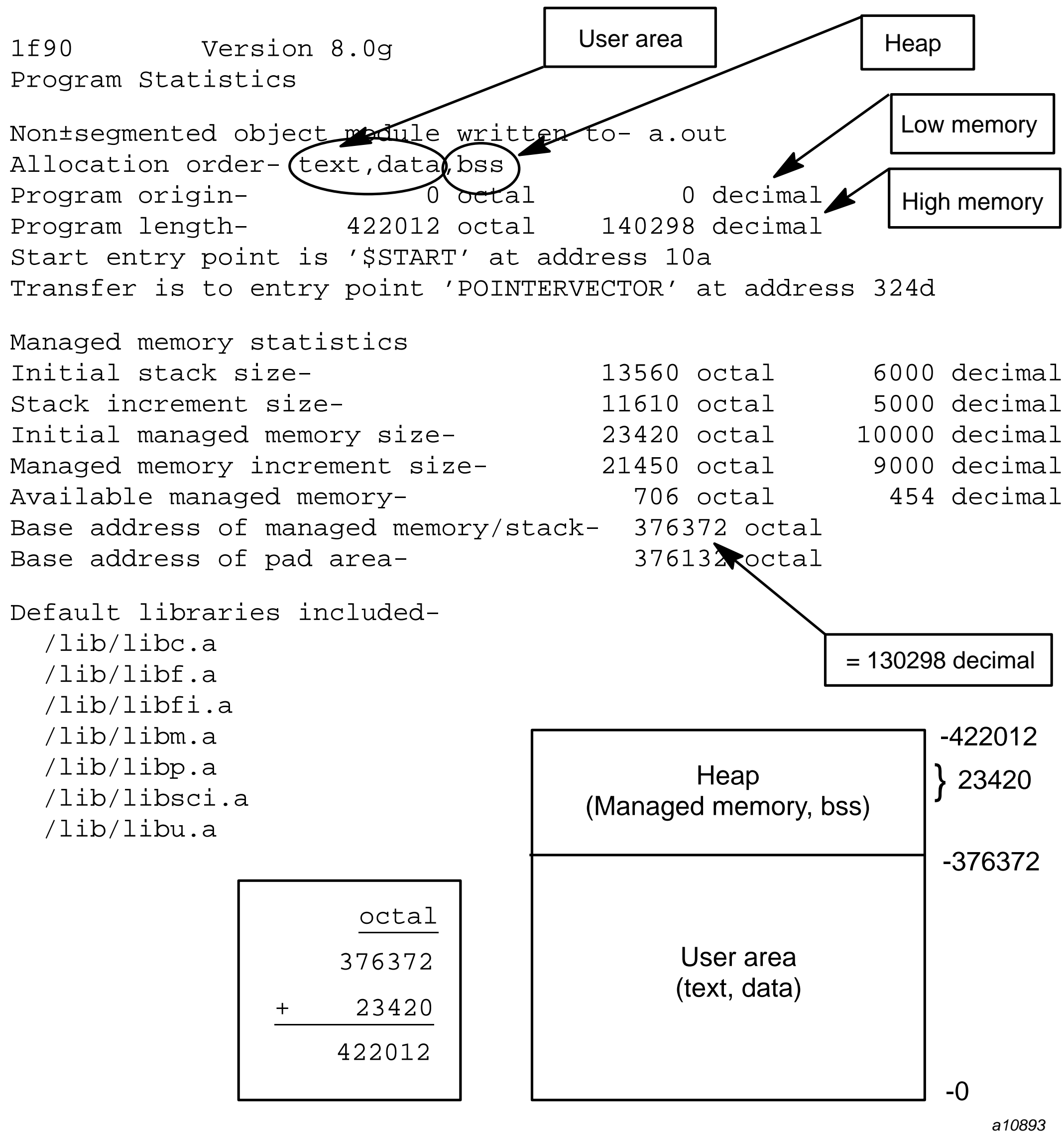


Figure 9. Load map statistics

Optimizing I/O [6]

This chapter describes the following techniques for optimizing I/O-bound code:

- Optimizing formatted I/O (Section 6.1, page 79)
- Optimizing large, sequential, unformatted I/O requests (Section 6.2, page 83)
- Optimizing small, sequential, unformatted I/O requests (Section 6.3, page 87)
- Optimizing for direct access I/O (Section 6.4, page 87)
- Optimizing asynchronous I/O (Section 6.5, page 89)
- Using an optimal storage device (Section 6.6, page 91)
- Minimizing system calls (Section 6.7, page 94)
- Using the SSD-I on Cray SV1ex systems (Section 6.8, page 94)

For more information on I/O, see the *Application Programmer's I/O Guide*.

6.1 Optimizing Formatted I/O

Formatted I/O is the slowest I/O and is useful only when the files must be viewed by people or transferred to systems other than Cray systems. However, if you are transferring the data to a system other than a Cray system, you can easily send the unformatted (binary) version instead of the formatted ASCII version by using the Cray foreign file conversion facility provided by the Flexible File I/O (FFIO) library. Use the techniques described in the following sections to optimize code that contains formatted I/O.

6.1.1 Changing to Unformatted I/O

If possible, change formatted I/O to unformatted I/O by using one of the following methods:

- In Fortran code, remove references to the `FORMAT` statement label and modify the Fortran `OPEN` statement to include `FORM='UNFORMATTED'`.

  Example:

  ```
  OPEN (10,FORM='UNFORMATTED')
  ```

- For C++ codes, `cout <<` and `cin >>` are formatted read and write member functions of the `iostream` class. Also, the `scanf`(3) and `printf`(3) function calls (including `fscanf`, `sscanf`, `fprintf`, and `sprintf`) require formatting to human-readable ASCII. Convert these functions to call unformatted I/O functions such as `fread` and `fwrite` instead. You can access FFIO by using the `ffread`(3) and `ffwrite`(3) functions in your code in conjunction with the `assign`(1) command.
- To access the I/O layers provided by the FFIO libraries, use the `-F` command-line option with the `assign` command. This will provide access to the automated foreign file conversion.

6.1.2 Reducing the Amount of Formatted I/O

If you cannot change formatted to unformatted I/O, reduce the quantity of formatted I/O. Show only small samples of the data by using the following techniques:

- Change the code to show final results instead of many intermediate results.
- Change the code to show a checksum instead of the data itself.
- If the program sends data to another computer system (or printer), revise the program so that only the final version of the data is formatted.
- If you need to view the data, consider shipping it unformatted to a graphics postprocessor.

6.1.3 Increasing Formatted I/O Efficiency for Fortran Programs

Use the methods in the following sections to increase formatted I/O efficiency for Fortran programs.

6.1.3.1 Minimizing the Number of Data Items in the I/O List

With the Cray Fortran Compiler you can increase formatted I/O efficiency by minimizing the number of data items in the I/O list. Consider the following example:

```
DIMENSION X(20), Y(10), Z(5,30)
WRITE (6,101) M, (X(I), I=1,20), Z(M,J)
```

With vectorization turned off, this `WRITE` statement represents 22 data items. In this case, the `WRITE` operation would require 22 calls to the library routines

that drive the `WRITE` statement. When vectorization is turned on, the compiler treats each innermost implicit `DO` loop as a single data item, so that the preceding `WRITE` statement requires only three calls.

If you rewrite the statement as follows, the parameter list always represents three calls, even if all optimization is turned off:

```
WRITE (6,101) M, X, Z(M,J)
```

6.1.3.2 Using a Single `READ`, `WRITE`, or `PRINT` Statement

To increase formatted I/O efficiency for Fortran programs, read or write as much data as possible with a single `READ`, `WRITE`, or `PRINT` statement. Consider the following example:

```
    DO J = 1, M
      DO I = 1, N
        WRITE (42, 100) X(I,J)
100     FORMAT (E25.15)
      ENDDO
    ENDDO
```

It is more efficient to write the entire array with a single `WRITE` statement, as follows:

```
    WRITE (42, 100) ((X(I,J),I=1,N),J=1,M)
100 FORMAT (E25.15)
```

The following statement is even more efficient:

```
    WRITE (42, 100) X
100 FORMAT (E25.15)
```

Each of these three code fragments produces exactly the same output; however, the latter two examples are about twice as fast as the first. Also, the latter two examples are equivalent only if the implied `DO` loops write out the entire array, in order, and without omitting any items. You can use the format to control how much data is written per record.

6.1.3.3 Using Longer Records

To increase formatted I/O efficiency for Fortran programs, use longer records if possible. Because a certain amount of processing work is necessary to read or write each record, it is better to write fewer long records, rather than more short records. Consider the following example:

```
    WRITE (42, 100) X
100 FORMAT (E25.15)
```

If you change it as follows, the resulting file will have 80% fewer records and, more importantly, the program will execute faster:

```
    WRITE (42, 101) X
101 FORMAT (5E25.15)
```

Be careful to ensure that the resulting file does not contain records that are too long for the intended application. For example, certain text editors and utilities cannot process lines that are longer than a predetermined limit. Generally, lines that are not longer than 128 characters are safe to use in most applications.

6.1.3.4 Using Repeated Edit Descriptors

To increase formatted I/O efficiency for Fortran programs, use repeated edit descriptors whenever possible. For integers that fit in 4 digits (that is, less than 10,000 and more than −1000), avoid the following format:

```
200 FORMAT (16(X,I4))
```

Instead, use a format of the following form:

```
201 FORMAT (16I5)
```

6.1.3.5 Using Data Edit Descriptors That Are the Same Width as the Character Data

To increase formatted I/O efficiency for Fortran programs, when reading and writing character data, use data edit descriptors that are the same width as the character data. For `CHARACTER*n` variables, the optimal data edit descriptor is `A` (or `A`*n*). For Hollerith data in integer variables, the optimal data edit descriptor is `A8` (or `R8`).

6.1.4 Increasing Formatted I/O Efficiency for C++ Programs

Calling a function increases overhead. To decrease overhead and increase formatted I/O efficiency for C++ programs, combine multiple calls to I/O functions into fewer calls. Consider the following example:

```
for (i=0; i<N;) {
   fprintf(o1,"%d ",a[i]);
   ++i;
   if (i%5 == 0) fprintf(o1,"\n");
```

```
}
```

If you change it as follows, the resulting code will make 80% fewer calls to `fprintf` and the program will execute faster:

```
for (i=0; i<N; i+=5) {
    fprintf(o2, "%d %d %d %d %d\n"
          a[i], a[i+1], a[i+2], a[i+3], a[i+4]);
}
```

6.1.5 Increasing Library Buffer Sizes for Formatted I/O Requests

For sequential-access formatted I/O files, the buffer size should be set equal to the length of a record or a multiple of that number. Generally, larger is better when buffering sequential access files.

To specify the library buffer size for Fortran, use the `assign`(1) command with the following options:

`assign -b` *size*

For C++, use the `setvbuf`(3C) library function.

6.2 Optimizing Large, Sequential, Unformatted I/O Requests

Sequential access indicates that data items in a file have an implicit order. Unless the code issues positioning requests such as `fseek`(3) or `rewind`(3), the system always accesses the next record automatically. If the code is issuing sequential, unformatted I/O requests larger than 1 MW, use the techniques described in the following sections to optimize its I/O.

6.2.1 Changing I/O File Format to Unbuffered and Unblocked

The default I/O file format for sequential unformatted Fortran I/O is COS blocked (`assign -s cos f:` *filename*), which means that the I/O request uses the library buffer and bypasses cache. Although the COS blocked file format helps fulfill the Fortran standard for sequential, unformatted I/O by marking (or blocking) record positions within a file, it is not the fastest I/O available for large, sequential transfers. COS blocked I/O requires user CPU time to create and insert (or interpret and remove) the control words.

If the code is issuing sequential, unformatted I/O requests larger than 1 MW (8 MB), change the I/O file format to unbuffered and unblocked by one of the following methods.

- Use the `-s u` option, as in the following example.

```
assign  -s u  f:filename
```

- Specify the FFIO `system`, or `syscall` layer, as shown in the following `assign`(1) command examples.

```
assign  -F system  f:filename
assign  -F syscall  f:filename
```

C++ codes can access the FFIO libraries by using the `ffread`(3C) and `ffwrite`(3C) I/O function calls in conjunction with the `assign` command.

Using the unbuffered, unblocked I/O file format requires you to construct *well-formed I/O requests* in the code. These are simply I/O requests that begin and end on disk sector boundaries, usually 512 words (4096 bytes) or a multiple of 512 words. This unit of measurement is also known as a *block* or *click*. See your system administrator to determine the sector size of the disks you are using.

6.2.2 Converting to Asynchronous I/O

Converting to asynchronous I/O is a way to continue I/O activity in parallel with the code's processor computation. If there are operations in the code that can be executed while the code is waiting for I/O to complete, convert the code to asynchronous I/O. For example, if the code contains any of the following sequences, converting to asynchronous I/O might reduce elapsed time:

- Repetitive patterns of input, computation on that data, output, then input again
- I/O that appears in a loop

Most prominent sequential, unformatted I/O requests that consume a majority of the code's elapsed time will benefit from code conversion to asynchronous I/O. You can convert to asynchronous I/O by using the `assign`(1) command or by modifying your source code.

6.2.2.1 Using the `assign` Command to Convert Code to Asynchronous I/O

The easiest way to convert code to asynchronous I/O is by using an FFIO layer, either `cachea` or `bufa`, with the `assign`(1) command, as follows:

```
assign  -F cachea:bs:nbufs  f:filename
assign  -F bufa:bs:nbufs  f:filename
```

The *bs* argument specifies the size in 512-word blocks of each cache page or buffer. The *nbufs* argument specifies the number of cache pages (or buffers) to use. You can tune these arguments to better suit the I/O activities of the code.

If the code requires the use of COS blocked format, you can establish a specialized FFIO layer to provide asynchronous access by using the following `assign` command:

```
assign  -F cos.async  f:filename
```

6.2.2.2 Optimizing Asynchronous I/O

You can modify the source code to take better advantage of the asynchronous FFIO layer by breaking up a large I/O request into smaller iterative requests. Within the iterations, perform the necessary computation on that data. This technique is called *double-buffering*.

With double-buffering, two sets of data (buffers) are active at any given moment for each stream of input or output data. One buffer is active in CPU work, while the other is active in I/O (reading or writing). In a typical double buffer scheme, the I/O and CPU work sets are staggered, as in the following algorithm:

1. The first set of input data is read.
2. The second set of input data is read while the processor works on the first set of input data.
3. The third set of input data is read while the processor works on the second set of input data and the first set of data is output.
4. This sequence continues until all data is read. As the last data set is read, the next-to-last processor work is in progress, and the third-from-last data set is output.
5. The processor works on the last data set and the next-to-last data set is output.
6. The final data set is output.

Example 1: C++ Example of Converting to Asynchronous I/O

In the following C++ example, the first input is the `reada`(2) system call in front of the `for` loop. Inside the loop, the first `recall`(2) system call synchronizes the previous `reada`, and the second `recall` synchronizes the previous `writea`. Notice that the `recall` system call is needed for synchronization. Generally, a second asynchronous system call to the same file descriptor will not block execution for the first. This is called *queuing asynchronous I/O*, because each asynchronous request enters an I/O queue without blocking execution.

```
#include <sys/types.h>
#include <sys/iosw.h>
#include <fcntl.h>
#define N 1001472
#define M 10
float a[N][2], b[N][2];
struct iosw sw[2], *prdsw[]={&sw[0]}, *pwrsw[]={&sw[1]};
int rfd, wfd, i, ird=0, iwk;
void work(float a[],float b[]);

main () {
  rfd = open ("infile", O_RAW | O_RDONLY);
  wfd = open ("outfile", O_RAW | O_CREAT | O_TRUNC | O_WRONLY, 0644);
  reada(rfd,(char *) &a[0][ird], N*sizeof(float), *prdsw, 0);
  for (i=0; i<M; i++) {
    iwk=ird;     ird=(ird+1)%2;
    recall (rfd, 1, prdsw);
   if (i != M-1)
      reada (rfd, (char *) &a[0][ird], N*sizeof(float), *prdsw, 0);
    work(&a[0][iwk], &b[0][iwk]);
    if (i != 0) recall (wfd, 1, pwrsw);
    writea (wfd,(char *) &b[0][iwk], N*sizeof(float), *pwrsw, 0);
  }
}
```

6.2.3 Using Effective Library Buffer Sizes

For large, sequential, unformatted I/O requests, enlarge the program's library buffer to at least the size of its largest record, if possible. To specify the library buffer size for Fortran, use the `assign`(1) command with the following options:

`assign -b` *size* `f:`*filename*

For C++, use the `setvbuf`(3) library function.

6.3 Optimizing Small, Sequential, Unformatted I/O Requests

If the code is issuing sequential, unformatted I/O requests that are 1 Mword or smaller, use the techniques described in the following sections to optimize I/O.

6.3.1 Using Effective Library Buffer Sizes

For small, sequential, unformatted I/O requests, use an effective library buffer size by ensuring that the library buffer is at least the size of the largest I/O request or a multiple of that size. For Fortran, use the `assign -b` *sz* command to specify the library buffer size. For C++, use the `setvbuf` library function.

6.3.2 Increasing I/O Request Size and Issuing Fewer Requests

To optimize small, sequential, unformatted I/O requests, increase the size of the I/O requests and issue fewer requests. This helps to reduce the overhead of both system and user CPU time and also may allow you to use the optimization techniques that apply to large I/O requests (see Section 6.2, page 83). You can use the following techniques to increase the size of the I/O requests:

- Read or write larger array sections instead of one element at a time.
- Combine read requests or write requests into a single read request or single write request.
- Extract I/O from inner loops.

6.3.3 Using the Memory-Resident (MR) FFIO Layer

For small, sequential, unformatted I/O requests, if the file called by the code is heavily reused, the memory-resident (MR) layer in FFIO can improve performance over disk I/O by allowing the first portion of the file to reside in memory. For information on the MR layer, see Section 6.6.1, page 92.

6.4 Optimizing Techniques for Direct Access I/O

Direct access indicates that a program can access records or data at any point in the file. This also can be called *nonsequential* or *random* access I/O.

6.4.1 Fortran Direct Access I/O

The Fortran standard provides two types of access: sequential and direct. Sequential access restricts the program to reading from or writing to the I/O unit with records of any length in sequential order. Direct access divides the file associated with the I/O unit into fixed-length records, and allows the program to read or write records randomly. You can achieve Fortran direct access by opening a file with the `ACCESS=DIRECT` keyword on the `OPEN` statement and specifying the fixed record size with the `RECL` keyword. All references to that file must specify the record number, `REC`, on subsequent `READ` and `WRITE` statements.

Example 2: Fortran Direct Access

```
OPEN (22,ACCESS='DIRECT',RECL=8000)
READ (22,REC=10) (DATA(I),I=1,1000)
WRITE (22,REC=2) (OUTNUM(J),J=1,150)
```

6.4.2 C++ Direct Access I/O

C++ programs do not use the I/O functions that transfer data to accomplish random access. C++ programs use the `fseek`(3) function or the `lseek`(2) system call to set the position in the file of the next input or output operation. The position is set in bytes, beginning at zero. Thus, C++ programmers are completely responsible for record keeping and indexing.

Example 3: C++ Direct Access

```
stream = fopen ("file","r+");

bytes_per_word = 8;
nwords = 1000;

lrec = bytes_per_word*nwords;
fseek (stream,9*lrec,SEEK_SET);
fread (&data,bytes_per_word,nwords,stream);
fseek (stream,1*lrec,SEEK_SET);
fwrite (&outnum,bytes_per_word,150,stream);
fseek (stream,  lrec - bytes_per_word*150,SEEK_CUR);
fread (&data,bytes_per_word,nwords,stream);
```

6.4.3 Optimizing Techniques for Direct Access Code

If the program is reading or writing files in direct access (as opposed to sequential access), you may be able to improve performance using the following techniques:

- Ensure that the files are in binary file format and that they bypass the system cache by using the `assign -s bin` command.
- Ensure that the code is not using formatted or COS blocked file formats.
- Set the library buffer size as close to the length of a record (request) as possible without going under the length. This minimizes unnecessary data transfers, which detract from the performance of random I/O.
- For small, random I/O requests, use a smaller library buffer than the default. Limit its size to the record length of the code. This might improve performance by avoiding excessive unused data movement when filling the unused portion of the buffer.
- For large I/O requests, the library buffer size should be set equal to the length of the fixed-size record (request). To specify the library buffer size for Fortran, use the `assign -b` *sz* command. For C++, use the `setvbuf`(3C) library function.
- If the code makes repeated references to the same place in the data file, a memory-resident (MR) buffer might help if it can include the most frequently used area of data. For information on the MR layer, see Section 6.6.1, page 92.
- If the code uses word-addressable data, you can transfer the data faster with a binary file format (using the `assign -s bin` command), which also bypasses the system cache and forces use of the `GETWA` and `PUTWA` I/O routines without changing the source code. The `GETWA` and `PUTWA` I/O routines are among the fastest types of random-access I/O on Cray SV1 series systems, but they place the burden of record keeping and indexing on you.
- Rearrange the data file so that the code can process it sequentially. Sequential I/O is usually faster than direct-access I/O. You might be able to use separate files to accomplish the same effect.

6.5 Optimizing Asynchronous I/O Requests

In most code, synchronous I/O is used more often than asynchronous I/O (also known as *raw I/O*). *Synchronous I/O* indicates that control is returned to the calling

program after all requested data is transferred. The I/O transfer runs serially with respect to the processor work.

Asynchronous I/O indicates that control is returned to the calling program after the I/O process has started, but before the I/O is completed. The I/O transfer runs in parallel with respect to the processor work. The user program continues executing at the same time the I/O operation is executing.

If the code is using asynchronous I/O, use the techniques described in the following sections. Some of these methods increase processor overhead but decrease total elapsed time if there is significant work to do during the I/O transfer.

6.5.1 Using Unblocked File Format

To optimize asynchronous I/O requests, use unblocked file format if the code does not need to backspace, position the file pointer, read partial records, and so on. You can improve asynchronous I/O performance moderately by eliminating the overhead associated with record marking or blocking. This can be done in several ways, depending on the type of I/O and certain other characteristics.

For example, the following `assign` statements specify the unblocked file structure:

```
assign -s unblocked f:filename
assign -s u f:filename
assign -s bin f:filename
```

6.5.2 Avoiding Cache

For asynchronous I/O, avoid using cache by specifying the `assign -s u` command. This allows the data to transfer directly between the user process and the actual device without a stopover (with synchronization) in cache.

6.5.3 Using Effective Library Buffer Sizes

If the program is using the default I/O file format for sequential unformatted Fortran I/O, which is COS blocked (with the `assign -F cos` command), to optimize asynchronous I/O requests, ensure that the largest record size is less than or equal to half the library buffer size. *COS blocked I/O file format* indicates that the I/O request uses the library buffer and bypasses cache.

Setting the library buffer size to an even number greater than 63 blocks causes COS blocked files to perform double-buffered asynchronous I/O by dividing the library buffer in half. When the library buffer size is an even number of disk sectors, each half of the buffer is well formed. Thus, I/O requests for either half-buffer do not need to be rerouted through cache.

You can change the buffer size by using the `assign`(1) command to specify a special FFIO layer, as follows:

```
assign -F cos.async:size f:filename
```

6.5.4 Balancing Workload

Device I/O speeds are typically slower than processor computation speeds by several orders of magnitude. If the code does not perform sufficient computation between I/O requests, it will spend most of its time waiting for I/O and lose the benefit of asynchronous I/O. Try to balance both the I/O activity and the computation involving its data by moving as much of the processor work as possible into the code that lies between asynchronous I/O requests.

6.5.5 Minimizing Required Synchronization

During asynchronous I/O processing, code reaches a synchronization point at which it has to wait for I/O completion before continuing. With an imbalance between processor and I/O activity, this causes extended I/O wait time and an idle processor. If this happens frequently, attempt to restructure the code to reduce required synchronization points.

6.5.6 Tune FFIO User Cache

If you are using asynchronous I/O through the `cachea`, `bufa`, or `cos.async` FFIO layers, you can adjust their sizes by using the `assign`(1) command. For complete information on controlling buffers and cache pages, see the *Application Programmer's I/O Guide*.

6.6 Using an Optimal Storage Device

If you have some flexibility with the storage devices your code uses, ensure that it uses the fastest devices available for the appropriate situations. The following sections describe storage devices and the situations in which they are best used.

6.6.1 Memory-Resident (MR) Files

Use MR files for small requests, heavily reused files, or for large files in which most of the I/O activity occurs at the beginning of the file. The `assign`(1) command provides an option to declare certain files to be memory resident. This option causes these files to reside within the field length of the user's process; its use can result in very fast access times.

To be most effective, this option should be used only with files that fit within the process's field length limit. A program with a fixed-length heap and memory-resident files might deplete memory during execution. Sufficient space for memory-resident files might exist, but might not exist for other runtime library allocations.

The MR layer lets you declare that a file should reside in memory. The MR layer tries to allocate a buffer large enough to hold the entire file.

To use the MR layer for Fortran code I/O, use the following `assign` command and then rerun the program. To use the MR layer for C++ code I/O, use the FFIO system I/O calls in the code first (`ffread`, `ffwrite`, and so on), and then use the `assign` command and rerun the program.

`assign -F mr:`*size* `f:` *filename*

The `-F` option invokes FFIO. The `mr` specification selects the memory-resident layer of FFIO. The *size* specification is the maximum size of the buffer in 512-word blocks. The *filename* specification is the name of the file.

6.6.2 Memory-Resident Predefined File Systems

Large memory systems might have predefined file systems resident in memory. Memory-resident file systems provide memory-to-memory speed, which is the fastest I/O available. Your system administrator can tell you which file systems are mounted in memory, and you might have access to create data files in those directories.

6.6.3 Disk Striping

If your file system is composed of partitions on more than one disk, using the disks at the same time can result in performance improvements. This technique is called *disk striping*. Disk striping can be accomplished through either hardware or software.

For example, if the file system spans three disks — partitions 0, 1, and 2 — it might be possible to increase performance by spreading the file over all three disks equally. Although 300 sequential writes might be required, assign only 100 to each disk and the disks can write simultaneously.

For *hardware striping*, your system administrator configures the disk and you can request that a file be opened on the striped disk during the `OPEN` system call.

Use *software striping* only for very large records, because all of the disk heads must do seeks on every transfer. Software striping can be useful for production jobs that monopolize the system resources, but it can impede total system throughput if used on a heavily loaded multiuser system.

To specify software striping, consult with your system administrator to identify disk partitions for your file system. You can also use the `df`(1) command with the `-p` option, but always consult your system administrator before attempting this technique. You can achieve software striping by placing the desired file system partitions in the `assign`(1) command, as shown in the following two examples:

```
assign -p 0-2  -n 300:48 -b 144  f:filename
assign -p 0:1:2 -n 300:48  -F cos:144  f:filename
```

Factors such as channel capacity might limit the benefit of striping. Disk space on each partition should be contiguous and preallocated for maximum benefit.

6.6.4 Disk Arrays

Using disk arrays such as DA-60 and DA-301 can be faster than single disk drives such as DD-60, DD-42, and DD-301.

6.6.5 Disks

If possible, use disks only for files that are accessed one or two times or for saved files that are read at a later time. Try to use memory for most other activity.

Large disk files on a multiuser system can easily become fragmented across different disk cylinders and cause increases in I/O wait time from the device. Fragmentation causes a higher number of requests for disk space allocation and more physical seek requests. You can examine disk file fragmentation with the `/etc/fck` command. For more information, see the `fck`(1) man page.

Avoid disk file fragmentation by preallocating disk space for the data files that will be stored there. In Fortran, you can use the `assign -n sz` command. In C++, use the `ialloc`(2) system call after the file is opened. If it is necessary to

allocate space prior to C++ program execution, use the `setf`(1) command. For more information, see the `setf`(1) man page.

The `assign`(1) and `setf`(1) commands will, by default, obtain contiguous allocation if it is available. If they do not succeed, you can force contiguous allocation by using the `-c` flag on either command, or by using the `IA_CONT` flag on the `ialloc`(2) system call.

6.6.6 Tapes

Consider tape to be a long-term storage device. Tape is both cost-effective and disaster-resistant. Before selecting tape, consider that it has slower access speed and that there is contention for the drive and delays for mounting. However, tape is appropriate for long-term archive storage of very large data files.

6.7 Minimizing System Calls

With few exceptions, system calls are required for all physical I/O requests and data movement to or from the library buffer. The following options minimize system calls:

- Ensure that I/O requests are as large as possible. For example, write whole arrays rather than one row at a time. Group multiple arrays into one write statement.
- Use larger buffers (or use cache) to capture many I/O requests in the user process space before the I/O library transfers the data out.
- Use scratch files for intermediate data that you no longer need after the code completes execution. This can eliminate unnecessary data movement and might avoid the device entirely.

  Scratch files are temporary and are deleted when they are closed. To create a Fortran scratch file, open a file with `STATUS='SCRATCH'` and use `STATUS='DELETE'`.
- Use the MR layer when appropriate (see Section 6.6.1, page 92).

6.8 Using the SSD-I on Cray SV1ex Model Systems

Cray SV1ex systems are equipped with an internal solid-state storage device (SSD-I). Using the SSD-I as a storage device for your data can improve the performance of I/O. Either 32 or 96 GB of fast storage are available on the

Cray SV1ex system. You can find out how much SSD-I space your system has by entering the `sysconf`(1) command. (The output in this example does not have a standard size SSD-I.)

```
HARDWARE: SERIAL= SN3398   MFTYPE= CRAY-SV1   MFSUBTYPE= SV1ex
          NCPU= 32   NSSP= 8   NMSP= 6   NIOM= 0   CPCYCLE= 10.0000 ns
          MEM= 4294704896   NBANKS= 2048   CHIPSZ= 268435456   XMEM= 25165824
          AVL= YES   BDM= YES   EMA= YES   HPM= YES   BMM= YES
          SSD= 12884901888   SSDRINGS= 7   IOS= MODEL_F   RINGS= 18   NCHAN= 8

SOFTWARE: RELEASE= 10.00   POSIX VERSION= 199009   SECURE SYS= ON
          SYSMEM= 33216512 WRDS   USRMEM= 4261488384 WRDS
          OS_HZ= 60   CLK_TCK= 100000000
          JOB_CONTROL= YES   SAVED_IDS= YES   SCTRACE= ON
          UID_MAX= 16777215   PID_MAX= 100000
          ARG_MAX= 49999   CHILD_MAX= 98   OPEN_MAX= 64
          NMOUNT= 150   NUSERS= 300   NPTY= 200
          NDISK= 16    SDS= 15703040    NBUF= 10000
          POSIX_PRIV= ON    SECURE_MLSDIR= SECURE    SECURE_MAC= OFF
          PRIV_SU= ON    PRIV_TFM= OFF
```

The SSD-I is essentially an extension of main memory that is physically located inside the mainframe. It is logically located between disk and main memory. Once your system administrator has configured part of the SSD-I to be available for user access as a secondary data segment (SDS), it is available to you for any of the following purposes:

- As a volatile pseudo disk/swap partition.
- For `ldcache`(8) use. It allocates part of the SSD-I as buffer space for I/O going to a file system.
- For SDS buffering; see the following description.

  There are two ways to use the SSD-I for buffering that will improve the access time for your program:

  – Make the file SSD-resident by entering:

    ```
    % assign -F sds:16385 mydata
    ```

    This statement sets up 16,385 blocks of SDS space for the file `mydata`.

– If the file is larger than the SDS space, you can essentially switch your FFIO buffers from memory to SDS space as follows:

```
% assign -F sds:32:32:4 myfile
```

Note: The `assign` command by default applies to Fortran I/O only. C programmers must rewrite their I/O to use FFIO library routines.

Data speeds for these examples can be up to 1.5 GBps. To approach the peak speed of 80 GBps, you would have to hand-code the computational arrays of the program to be properly aligned in physical memory and directly use `SSREAD`(2) and `SSWRITE` system calls to read and write those arrays.

An advantage of using the SSD-I is that your data will remain there during your program's run. It will not be removed, even if your program is swapped out, until your program completes.

Glossary

alias

The `alias` shell command lets you define a synonym (a convenient name) for a command or command string. It lets you define a more mnemonic name for an existing command or a shorthand for a longer command string.

Autotasking

A trademarked process of Cray that automatically divides a program into individual tasks and organizes them to make the most efficient use of the computer hardware.

cache

A kind of temporary memory that speeds up load and store operations between memory and the registers of a processor.

chaining

A process of linking instructions together to save register storage time. Each instruction passes its results to the next linked instruction so that several operations may be done in approximately the same amount of time as one operation.

This function allows a vector register that is being used as a result register in one instruction also to be used as an operand register in a following instruction. By chaining vector instructions, overall speed is greatly increased; as soon as the first vector instruction has completed the function on element 0, that result is available to the second vector instruction as an operand. The first vector instruction does not have to complete processing on all vector elements before the second vector instruction can start processing.

chime

A sequence of vector operations that can be chained into a single pipeline. The limitation on such a sequence is that the same vector functional unit cannot be used twice in the same chain. Therefore, a loop that contains two vector adds, for example, contains at least two chimes because there is only one vector add functional unit.

data dependence

Occurs when the data that results from one segment of code depends on the data that results from previous segments of code.

disk striping

Multiplexing or interleaving a disk file across two or more disk drives to enhance I/O performance. The performance gain is function of the number of drives and channels used.

double-buffering

Modifying source code with a technique by which you break up a large I/O request into smaller iterative requests. Within the iterations, you perform the necessary computations on that data.

FLOPS

Floating-point operations per second.

gather/scatter

Either collecting data from multiple processors to a single processor (gather) or dispersing data from a single processor to multiple processors (scatter). The Fortran compiler multistreams an ordered scatter (a scatter that involves constant, as opposed to random, strides through the array being scattered).

heap

A section of memory within the user job area that provides a capability for dynamic allocation. See the heap memory management routines in the library documentation.

induction variable

A variable in a loop that controls the execution of that loop (for example, in Fortran, `I` is the induction variable in `DO I = 1, 100`; in C++, `i` is the induction variable in `for (i=0;i<100;i++)`).

inlining

The process of replacing a user subroutine or function call with the subroutine or function itself. This saves subprogram call overhead and may allow better

optimization of the inlined code. If all calls within a loop are inlined, the loop becomes a candidate for vectorization, tasking, and multistreaming.

load balancing

A process that ensures that each processor involved in a program performs roughly equal work.

loop counter

An integer variable that is incremented or decremented by an integer constant expression on each pass through the loop.

loop fusing

A code optimization technique by which two independent loops with the same iteration count are combined into one vector loop.

loop pushing

A code optimization technique by which a loop containing a subprogram call is moved into the called subprogram.

loop splitting

A code optimization technique by which a loop that contains both vectorizable work and scalar work is split into two loops: one that vectorizes, and one that does not.

loop unrolling

A code optimization technique in which the statements within a loop are replicated while reducing the trips through the loop.

loop unwinding

A code optimization technique in which a loop is unrolled completely, so that it is no longer a loop.

memory bound

A program that is no longer able to execute because it has run out of room in memory.

memory management

Management that allows memory to be allocated dynamically to programs while they are executing.

MSP

A multistreaming processor capable of performing *multistreaming* operations. It consists of four SSPs (single-streaming processors).

multitasking

An optimization method that incorporates multiple interconnected processors; these processors each run a part of a program simultaneously (in parallel) and share resources such as memory, storage devices, and printers. This term is used interchangeably with *parallel processing*.

multistreaming

An optimization technique that automatically schedules one or more Cray SV1 multistreaming processors (*MSPs*) for a program and divides the work for each MSP among four single-streaming processors (SSPs).

parallel processing

Processing in which multiple processors work on a single application simultaneously.

parallel region

An area within a program which multiple processors can productively execute in parallel. Its opposite is a *serial region*.

pipelining

A method of executing a sequence of instructions in a single processor so that subsequent instructions in the sequence can begin execution before previous instructions complete execution. This assembly-line approach to processing instructions is also called instruction pipelining or hardware pipelining.

private data

Data that is replicated on as many processors as define that data, rather than spreading one copy of the data over all processors.

raw I/O

A method of performing input/output in which the programmer must handle all of the I/O control. This is basically unformatted I/O.

redundant code

Code contained within a parallel region that is executed by all associated tasks, using the same data and generating the same results.

scalar

(1) In Fortran, a single object of any intrinsic or derived type. A structure is scalar even if it has a component that is an array. The rank of a scalar is 0. (2) In C and C++, integral, floating, and pointer types are collectively called scalar types. (3) A nonvectorized, single numerical value that represents one aspect of a physical quantity and may be represented on a scale as a point. This term often refers to a floating-point or integer computation that is not vectorized; more generally, it also refers to logical and conditional (jump) computation.

scalar processing

A sequential operation in which one instruction produces one result; it starts an instruction, handles one operand or operand pair, and produces one result. Scalar processing complements vector processing by providing solutions to problems not readily adaptable to vector techniques.

scalar temporary

A simple variable defined and later referenced during each pass through a loop; it is not referenced outside the loop. The compiler can either replace a scalar temporary with a register, or it can eliminate it.

stack

(1) A data structure that provides a dynamic, sequential data list that can be accessed from either end; a last-in, first-out (push down, pop up) stack is accessed from just one end. (2) A dynamic area of memory used to hold information temporarily; a push/pop method of adding and retrieving information is used. (3) A portion of computer memory and/or registers used to hold information temporarily. The stack consists of stack frames that hold return locations for called routines, routine arguments, local variables, and saved registers.

stack thrashing

Frequent stack expansion (overflow) and contraction (underflow).

stride, constant

An interval that is the same for all consecutive elements of a vector. On the Cray SV1 system, vectorization requires a constant, odd-numbered stride to perform at peak efficiency. An array processed with a stride of 1, such as `A(1), A(2), A(3),` ... is efficient. Avoid powers of 2, such as `B(2), B(4), B(6),` ... . A stride that is not constant is illustrated by a sequence such as `A(1), A(2), A(3), A(5), A(8), A(13)`.

tailgating

Writing to a V register that is still being read from a prior vector instruction.

user area

A location at which the object code (text area) and external and static variables (data area) reside in memory.

user CPU time

Time accumulated by a user process when the process is attached to a CPU and executing. It is a fraction of *wall–clock* time, which measure the time from when you begin the execution of a program until execution completes. Wall-clock time includes the time a program is waiting for a CPU.

user process

An executable file becomes a user process after it is compiled and loaded. Your code is a UNICOS user process when it is executing.

vector

A computer vector is an array of numbers on which instructions operate; this can be an array or any subset of an array (such as a row, column, or diagonal). When arithmetic, logical, or memory operations are applied to vectors, it is referred to as vector processing.

vector array reference

An array element reference whose subscript expression is not a loop invariant. It is an array that is processed in vector registers.

vector length

The number of elements in a vector.

vector processing

A technique whereby iterative operations are performed on sets of ordered data. It provides results at rates exceeding the result rates of conventional scalar processing.

vector register

A vector (V) register is used for vector operations; successive elements from a V register enter a functional unit in successive clock periods.

vectorizable expression

An arithmetic or logical expression that consists of a combination of loop invariants, loop counters, vector array references, scalar temporaries, or a function with a vector version that has a vectorizable expression as an argument. This includes most Fortran intrinsic functions.

vectorizable loop

A loop that contains only vectorizable expressions (that is, expressions for which the compiler can produce vector code).

vectorization

Uses one instruction for the simultaneous performance of iterative operations on elements in sets of ordered data. It provides results at rates greatly exceeding those for conventional scalar processing, which works on only one element at a time.

vectorized loop

A source code loop that is processed with hardware vector registers.

well-formed I/O requests

I/O requests that begin and end on disk sector boundaries, usually 512 words (4096 bytes) or a multiple thereof.

Index

A
ALLOCATE keyword, 73
analyzing
  memory-bound code, 71
arrays
  disk, 93
arrays, private
  in multistreaming, 29
assign command
  accessing layers provided by FFIO libraries, 80
  bypassing system cache, 89
  COS-blocked format, 83, 90
  for library buffer size, 87, 89
  setting buffer size, 83
  specifying file format, 89–90
  specifying library buffer size, 86
  specifying MR layer, 92
  to avoid cache, 90
  to convert to asynchronous I/O, 85
  to invoke FFIO, 92
asynchronous I/O
  converting to, 84
  optimizing, 89

B
bit matrix multiply and multistreaming, 30
BMM and multistreaming, 30

C
C and C++
  multistreaming options, 27
cache
  avoiding, 90
chaining
  vector units, 8
code evaluation
  determining available user memory, 22
  initial, 11
  using hpm, 13
  using procstat, 71
  using the ja utility, 20, 24
computational intensity
  determining, 16
CPU time
  large amounts, 71
cpu(8) command
  with a multitasking job, 32
CPU-bound code
  definition, 11
  determining whether code is, 11

D
data edit descriptors, 82
data items in I/O list
  minimizing, 80
DEALLOCATE keyword, 73
delete keyword, 73
direct access I/O
  definition, 87
directives
  multistreaming, 27
disk arrays, 93
disk striping, 92
disk use, 93
double buffering, 85
dynamic common blocks memory
  management, 70
dynamic heap memory management, 69
dynamic memory
  alternatives, 72
  management types, 68

E
edit descriptors
  repeated, 82
elapsed time

definition, 2
evaluating code, 11
examples
direct access I/O, 88
heap initialization, 75
procstat, 71
setting initial heap size, 74

F

FFIO cache tuning, 91
FFIO libraries
accessing, 83
file format
unblocked for asynchronous I/O, 90
unbuffered and unblocked, 83
file systems
predefined memory resident, 92
files
memory resident, 92
flowcharts
initial evaluation, 11
optimization overview, 4
formatted I/O
increasing efficiency, 80, 82
increasing library buffer sizes, 83
optimizing, 79
reducing amount of, 80
Fortran
`IVDEP` directive, 46
multistreaming option, 26

G

GigaRing I/O
description, 9

H

-h stream option
for C and C++, 27
hardware
overview, 5
hardware expectations
single CPU, 14
hardware performance monitor (HPM), 13
heap definition, 69
heap initialization
example, 75
heap size
optimal, 75
help
from Cray, 3
HPM, 13
hpm report
sample, 14
hpm utility
using, 13

I

I/O
asynchronous, 89
asynchronous conversion, 84
changing to unformatted, 79
increasing request size, 87
large requests, 83
minimizing number of data list items, 80
optimizing for small requests, 87
optimizing formatted, 79
optimizing unformatted, 83, 87
unbuffered, unblocked format, 83
I/O-bound code
definition, 24
determining whether code is, 24
optimizing, 79
instruction buffer fetches, 15

J

ja report
checking for I/O-bound code, 24
checking for memory-bound code, 21
ja utility
using, 20

L

layer
memory resident, 87

library buffer sizes, 86
for asynchronous I/O, 90
for formatted I/O, 83
for unformatted I/O, 87
Load Map Program Statistics report
creating, 75
sample, 77
loader directives, 74
loop iterations
dividing among processors, 25
loopmark listing
creating, 41
loop type codes, 42

M

memory
using efficient storage for, 92
memory high water mark, 21
memory initialization, 74
memory management
conditions causing excessive system time, 70
dynamic common blocks, 70
dynamic heap, 69
hidden, 70
problems, 67
types, 68
memory size, 5
memory wait time
large amounts, 71
memory-bound code, 67
definition, 11, 19
memory-resident
files, 92
layer, 87
predefined file systems, 92
MFLOPS, 14
MIPS, 14
MR layer, 87
_MsBarrier routine
in multistreaming, 34
MSP
multistreaming processor, 25
MSP_STATS environment variable, 35
multistreaming
analysis, 16
and bit matrix multiply, 30
directives, 27
enabling, 26
in a multitasking program, 31
nested loops with, 29
on Cray SV1 systems, 25
options and directives, 26
performance analysis tools, 33
private arrays, 29
processor, 25
statistics in C and C++, 27
types of codes optimized, 29
versus multitasking, 31
with multitasking, 31
multitasking
with multistreaming, 31

N

NCPUS environment variable
setting number of MSPs, 26
nested loops
which is multistreamed, 29
new keyword, 73
nostream directive, 28

O

-O stream option
for Fortran, 26
optimizing code
autotasking, 56
computed safe vector length, 46
conditionally vectorized, 44
I/O bound, 79
initial evaluation, 11
loop blocking, 57
loop collapse, 52
loop fusion, 54
loop interchange, 51
loop unrolling, 49

outer-loop vectorization, 49
partially vectorized loops, 43
pattern matching, 55
processor performance, 13
reduction loop, 44
short loop, 45
streaming, 56
vector update, 45
vectorization inhibitors, 47
optimizing memory, 67
overview
optimization process, 2

P
performance analysis tools
with multistreaming, 33
performance measurement with multistreaming, 35
preallocation
disk space, 93
predefined file systems
memory resident, 92
preferstream directive, 28
PRINT statements, 81
process monitoring with procstat, 71
processor, 8
processor performance
multiple, 1
single, 1
procstat utility
description, 71
prof command
with multistreaming, 33
profview command
with multistreaming, 34

R
READ statements, 81
records
using longer, 81
reduction loop, 44
reusing heap space, 73

S
sar command, 22
SBREAK library routine, 71
scratch files, 94
SEGLDR
to specify initial heap size, 74
to specify optimal heap size, 75
sequential access
definition, 83
sequential I/O
optimizing, 83, 87
setvbuf routine
setting buffer size, 83
single processor
determining whether code is optimized for, 13
hardware expectations, 14
optimization method, 1
software striping, 93
SSP
single-streaming processor, 25
stack area, definition, 69
stack frames
definition, 70
stack thrashing, 70, 73
STOP statement, 73
storage devices
using optimal, 91
stream directive, 28
striping
disk, 92
software, 93
summation loop, 44
synchronization reduction, 91
sysconf command, 22, 25
system calls, minimizing, 94
system CPU seconds, 22
system CPU time
large amounts, 71

T
tailgaiting
vector units, 8

tape use, 94
target command, 22
threshold argument, 29
tools
  hpm, 13
  ja, 20, 24
  procstat, 71
  procview, 71

U

unblocked file format
  for asynchronous I/O, 90
unblocked I/O format, 83
unbuffered I/O format, 83
unformatted I/O
  changing to, 79
  optimizing, 83, 87
user area, definition, 69
user CPU seconds, 22
user CPU time
  definition, 2
user memory available, 22
user process, 69

V

vector collapse, 44
vector unit
  peak rate, 8
vectorization
  and multistreaming, 33
  dependencies, 47
  factors inhibiting, 47
vectorization overview, 39

W

waiting at barriers
  when multistreaming, 34
workload balancing, 91
WRITE statements, 81