

Delivering Performance on Sun: Optimizing Applications for Solaris

Technical White Paper



© 1997-2000 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, California 94303 U.S.A

All rights reserved. This product and related documentation is protected by copyright and distributed under licenses restricting its use, copying, distribution and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX Systems Laboratories, Inc. and the University of California, respectively. Third party font software in this product is protected by copyright and licensed from Sun's Font Suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Sun WorkShop, and Sun Enterprise are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UltraSPARC, SPARCCompiler, SPARCstation, SPARCserver, microSPARC, and SuperSPARC are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



Please
Recycle



Adobe PostScript

Contents

1.	Introduction	1
2.	Performance: Sun's Tools for Optimizing Applications	7
	Forte Developer Compilers — Optimized for Performance	8
	Highly Optimizing, Automatically Parallelizing Compilers	9
	Fortran and C: Autoparallelizing and Pragma Assisted	10
	Sun Performance Library	15
	Math Libraries	17
	Controlling the Floating-Point Environment	19
	mediaLib and the Visual Instruction Set	21
	Optimization Levels	22
	Optimizing for Speed — The -fast Option	23
	Profile Feedback	24
	Performance Tools	25
	Forte Developer Performance Analyzer	25
	Call Graph Profiles and gprof	33
	Multiplatform Support — 32-bit and 64-bit Applications	34
	Specify Processors — xarch	35
	UltraSPARC Compilers: Built for Speed	36
	Identifying Target Systems — xtarget	36
	Specifying Code Address Space — xcode	37

	Specifying Cache Properties — xcache	38
	Insert Padding — pad	39
	xsafe=mem	40
	xmemalign	40
	xprefetch	40
	Sun's Guidelines for Achieving Maximum Application Performance	41
	Uniprocessor Environments	41
	Multiprocessor Environments	43
	Common Pitfalls	45
	Tuning Interval Arithmetic Applications	48
3.	Quality: First Class Error Detection for Single and Multi-CPU Systems	49
	Multithreading dbx Functionality	49
	Batch Debugging	51
	Runtime Checking	52
	Memory Monitor	53
	Global Program Checking	54
	Analyzing C Source Code — LockLint	56
4.	Productivity: Intuitive, Efficient, and Extensible Application Development Using Forte Developer Products	57
	Integrated Programming Environment	58
	Integrated Editors	59
	Project Manager	60
	Data Visualization	60
	Fix and Continue	61
	Incremental Linker	62
	Forte C++ Enterprise and Personal Editions	62
	Integrated make Utility — Building Window	64
	Remote Monitoring	64
	Accessing Sun Documents On-line	65
5.	Summary	67
A.	Glossary	69
	References	73

Organizations today require high performance, high quality applications. To meet the needs of their application users, developers need a rich and integrated set of software development tools, along with high speed computers and networks. Modern tools enable many new programming techniques and use innovative technologies to improve the speed with which one can develop and run applications. Using the correct compiler options and optimizations can substantially improve performance without compromising stability.

This document focuses on how to use the right tools and employ the most effective techniques for delivering high performance applications on Sun™ workstations and servers. Using these techniques on Sun SPARC™ systems will enable developers to provide optimal solutions to today's mission-critical business problems. Specific techniques for optimizing applications for UltraSPARC™ platforms are also discussed.

Today's software developers need to:

- Write fast code (performance)
- Write correct code (quality)
- Write code fast (productivity)

Sun's Forte Developer 6 suite excels in meeting these three requirements by providing developers with everything they need to create high quality, high performance applications in a productive environment (Figure 1-1). In addition, Forte Developer 6 incorporates full support for 64-bit computing, including new compiler options, updated libraries, large file and address space support, and other related directives and options.

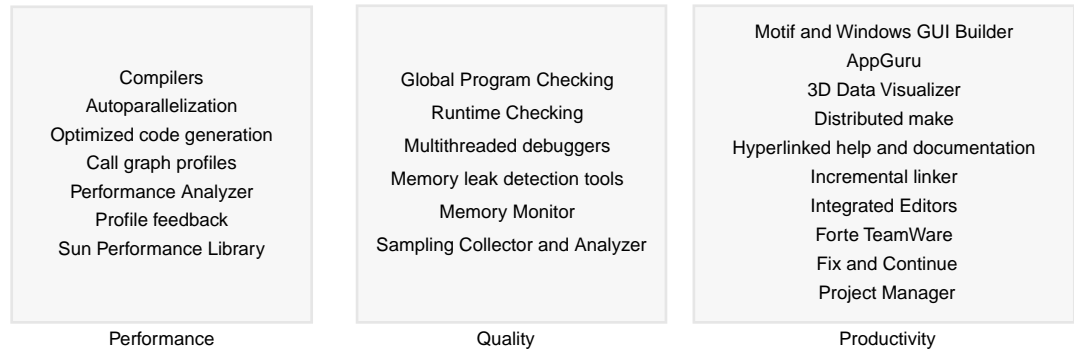


Figure 1-1 Sun's Forte Developer 6 suite provides all the tools developers need to develop high quality, high performance applications in a highly integrated environment

Performance

Forte Developer tools aid programmers in creating efficient applications that are portable across the entire Sun product family. Utilizing the extensive set of 32- and 64-bit options available in Sun's Forte compilers, applications can be built to meet a broad range of needs. For instance, an application can be compiled to provide reasonable performance on any SPARC processor, resulting in a single application binary for all Sun systems. Or, using different compiler options, the same application can take advantage of features found in the latest 64-bit UltraSPARC processor to run with maximum performance. Using the same application source code, optimized application binaries can be created for each SPARC architecture, maximizing performance across the entire SPARC product line.

Sun, a leading vendor of symmetric multiprocessing systems, also provides an excellent environment for running parallelized applications. Forte Developer 6 allows programmers to take advantage of this environment with tools that simplify the task of explicit parallelization. Or, if appropriate, Forte Developer 6 can automatically parallelize C and Fortran programs. Chapter 2, *Performance: Sun's Tools for Optimizing Applications*, describes Forte Developer's tools for creating optimized and parallelized applications and suggests ways to use them effectively.

Quality

Software developers need sophisticated debugging and analysis tools that enable the construction of quality solutions to meet today's business needs. Studies have shown that the earlier a software defect is found in the development cycle, the less time consuming and therefore less costly it is to fix.

Forte Developer 6 offers interactive debugging tools which enable a developer to run a program in a controlled fashion and to investigate its state during execution. Forte Developer 6 also includes runtime checking, a feature which detects memory access errors and memory leaks — common sources of code performance degradation. By catching these errors during application development, programmers avoid application performance degradation that can lead eventually to application crashes during deployment. Chapter 3, *Quality: First Class Error Detection for Single and Multi-CPU Systems*, describes Forte Developer's facilities for building quality into optimized and parallelized applications.

Productivity

To be as productive as possible, software developers need an integrated development environment of compilers, debuggers and other tools. They need to be able to work on tasks concurrently, as well as to collaborate with other developers. Their tools also need to be capable of developing optimized and parallelized applications with peak deployment performance.

Sun systems and software aid developers in two key aspects of software development: first, the development and testing of an application, and second, the application's portability and performance. To facilitate development and testing, Sun's development products run on all SPARC platforms, enabling developers to work effectively on any machine. The Solaris™ operating environment is also binary compatible across systems and from release to release, helping to protect system investment. Built for networking, Sun desktops and servers support efficient team development and provide easy access to a variety of testing tools. Indeed, Sun's high performance CPUs, memory and I/O systems all contribute to reducing development cycles and enable developers to work on concurrent tasks, thereby increasing productivity.

Chapter 4, *Productivity: Intuitive, Efficient, and Extensible Application Development Using Forte Developer Products*, describes the productivity features of Forte Developer 6.

The Forte Developer 6 Product Family

The Forte Developer 6 product family is an award-winning, powerful, visual application development environment that incorporates all the tools developers need to create today's enterprise software, including client-server, object-oriented, and mission-critical single and multi-threaded applications. Forte Developer 6 integrates a wide variety of tools into a productive development environment:

- Suite of optimized compilers for C, C++, FORTRAN 77, Fortran 90 and Fortran 95
- FORTRAN 77 - Fortran 90 interoperability
- Support for ANSI C++ and the C++ Standard Library
- Intuitive graphical user interface
- Source browser
- Project Manager
- Three dimensional data visualizer
- Highly optimized mathematical libraries
- Sophisticated MT-aware debugger
- Runtime checker
- Performance analyzer
- Software configuration management tools

The Forte Developer 6 development tools are available in several editions:

- *Forte C*, a suite containing a C compiler, debugger, and performance tools
- *Forte C++ Personal Edition 6*, which is node-lock licensed, for parallelized C++ applications
- *Forte C++ Enterprise Edition 6*, which is licensed by FlexLM, for parallelized C++ applications
- *Forte Fortran Desktop Edition 6*, which is node-lock licensed, for single-threaded Fortran applications that do not need automatically parallelized or explicit, directive-based parallel code
- *Forte for High Performance Computing*, which is licensed by FlexLM, for clustered environments and multi-threaded Fortran applications needing automatically parallelized or explicit, directive-based parallel code

New Features in Forte Developer 6

The most advanced development suite from Sun to date, Forte Developer 6 has been designed to address the performance, quality, and productivity requirements of 64-bit computing environments. With this latest release, Forte Developer provides a variety of new and updated features, including:

- *Enhanced ANSI/ISO C++ compliance*, ensuring application portability across computing environments
- *New memory alignment options*, including general data element alignment (`-xmemalign`) and common block alignment on specified byte boundaries (`-aligncommon`)
- *OpenMP parallelization directives*, enabling developers to dictate how parallelization occurs in Fortran applications (`-mp=openmp`)
- *Relaxed debug compiling restrictions*, enabling developers to compile at `-O4` and `-O5` and/or any of the parallelization flags (`-xparallel`, `-explicitpar`, `-autopar`) with debugging (`-g`)
- *Expanded prefetch options*, enabling developers to force the generation of prefetch instructions on UltraSPARC platforms via explicit pragma prefetch directives (`-xprefetch`)
- *Enhanced array optimizations*, enabling aggressive array optimization to be performed at levels `-O4` and `-O5`
- *Support for interval arithmetic*, enabling Fortran 95 applications to exploit interval arithmetic for more reliable and predictable results (`-xia` and `-xinterval`)
- *New Sun Performance Library routines*, including subroutines for sparse matrices, convolution and correlation in one and two dimensions, complex vector FFTs, and Fourier transforms in two and three dimensions
- *UltraSPARC-III processor support*, including applications to take full advantage of the latest processor advances
- *Easier-to-use programming environment*, facilitating development and debugging efforts
- *Enhanced Performance Analysis tool*, enabling developers to analyze programs
- *New NEdit and Vim text editors*, increasing ease-of-use and improving developer productivity
- *New Balloon Expression Evaluator*, enabling developers to see the current values of expressions in a text editor
- *New Project Manager*, facilitating the coordination of files, programs, and targets associated with a development project

Sun's Commitment to Developers

Sun is constantly looking to, and shaping the future of computing by investing in new technology. Sun recognizes that a consistent, continuous application of resources is needed to meet the needs of a rapidly-changing computing marketplace. Significant investments in high performance, low-cost compilers, integrated environments, advanced debugging tools, and adherence to standards ensure that Sun customers will always have access to the best products available. Sun also pursues alliances with other industry trendsetters in a concerted effort to deliver new products and technologies that foster greater productivity, higher quality products, improved time-to-market, and ultimately an improved bottom line.

Forte Developer and UltraSPARC: A Powerful Combination

Today's businesses demand that programmers quickly produce high quality, high performance applications. Forte Developer 6 gives developers an integrated, powerful, visual environment that meets their performance, quality and productivity needs. When Forte Developer is combined with an UltraSPARC hardware platform, developers have the high powered development solution they need. And for maximum performance in application deployment, UltraSPARC platforms provide organizations with the powerful, high quality, high performance systems they need to be leaders in their markets.

Performance: Sun's Tools for Optimizing Applications

Programmers need high performance compilers and tools to successfully develop high performance 32- and 64-bit applications. The Forte Developer 6 family of products includes highly optimizing, automatically parallelizing compilers, libraries of highly optimized routines, as well as tools to help analyze and isolate code that can be tuned for additional runtime performance. Indeed, the proper use of algorithms, compiler options, library routines, and coding practices can bring significant performance gains. Developing with Forte Developer 6 compilers ensures programmers that their applications are optimized and parallelized to provide peak performance.

When developing high performance applications, programmers must select the right combination of compiler options, optimized libraries, and coding techniques. For a quick gauge of application performance, and to obtain maximum performance gain without code restructuring, developers can try the suggested compilation options in Figure 2-1. For a thorough understanding of the options employed and their effect on application performance and binary compatibility, developers should take care to read the remainder of this chapter.

Options for high performance 32-bit applications	Options for high performance 64-bit applications
% cc -fast -xtarget=ultra -xarch=v8plusa -xdepend example.c -lmopt -o example	% cc -fast -xtarget=ultra -xarch=v9a -xdepend example.c -lmopt -o example
% f90 -fast -xtarget=ultra -xarch=v8plusa example.f -o example	% f90 -fast -xtarget=ultra -xarch=v9a example.f -o example

Figure 2-1 A suggested approach to achieving maximum performance gains without code restructuring. Using these options, developers can compile applications to take maximum advantage of 32-bit and 64-bit UltraSPARC-based systems at the expense of application binary compatibility with other systems.

Forte Developer 6 Compilers — Optimized for Performance

Optimizing compilers are designed to provide significant improvements in program execution speed by using a variety of techniques to produce the most efficient sequence of machine instructions. Forte Developer compilers employ traditional as well as the latest in compiler optimization technology to produce fast, efficient, and reliable code:

- *Common subexpression elimination*, for expressions used multiple times. The compiler evaluates them once and saves the resultant value for later use, eliminating the necessity to recalculate.
- *Register allocation*, to make maximal use of processor registers when evaluating expressions and storing values.
- *Loop-invariant hoisting*, to evaluate expressions within a loop whose values do not change across loop iterations before loop begins execution.
- *Strength reduction*, to replace slower operations with faster ones. Typical cases include replacing a multiply operation inside a loop with an addition operation, and replacing a constant multiplier with shift and add operations.
- *Dead and redundant code elimination*, to remove computations that cannot be reached or that are redundant, decreasing code size.
- *Loop pipelining/unrolling*, to arrange loop code to ensure idle time and idle resources during one loop iteration may be applied to another.
- *Instruction scheduling*, to arrange the order in which instructions are executed and make optimal use of available machine resources.
- *Inlining*, to replace subroutine calls with their bodies. This reduces overhead and takes advantage of special circumstances, such as constant parameters.
- *Code motion*, to move instructions from frequently executed code segments to those less frequently executed.
- *Profile feedback*, to obtain frequency information about a program. The program is executed and the frequency information is applied to optimizations such as code motion and inlining.
- *Tail recursion elimination*, to convert recursive programs into iterative programs. This reduces overhead and saves stack space.
- *Loop parallelization*, to rearrange loop code so that multiple processors may work in parallel to complete the loop.

- *Cache blocking*, to rearrange loop code to make maximum use of the processor cache.
- *Loop interchange*, to reverse the order of nested loops to gain the advantages of improved loop parallelization or better cache blocking.

These and other optimization techniques have resulted in improved performance. For example, on an UltraSPARC platform there was a 5 percent increase in SPECint95 performance and a 14 percent improvement in SPECfp95 performance from Sun WorkShop 5.0 to Forte Developer 6.

Forte Developer compilers are available for C, C++, FORTRAN 77, and Fortran 90. Additional programming tools provide a highly-productive programming environment for the development of sophisticated software applications, offering a complete set of utilities for editing, compiling, linking, debugging, and tuning applications.

Highly Optimizing, Automatically Parallelizing Compilers

Parallelizing (or multithreading) an application recasts the compiled program to take full advantage of a multiprocessor system. Parallelization enables single tasks, such as loops, to run over multiple processors with a potentially significant increase in execution speed. Application programs can be multithreaded to enhance their efficiency on multiprocessor systems. Tasks that can be performed in parallel can be identified and reprogrammed to distribute their computations.

Sun compilers can automatically generate multithreaded object code that runs on multiprocessor systems. The compilers focus on loops as the primary language element supporting parallelism. Parallelization distributes the computational work of a loop over several processors without requiring modifications to the source program. The choice of which loops to parallelize and how they should be distributed can be left entirely up to the compiler, determined explicitly by the programmer with source code directives, or done in combination.

Sun's automatically parallelizing compilers are available for C and Fortran. These compilers have been architected to ensure that each provides the highest levels of correctness, debugging, tuning and performance.

Toward this end, Sun engineered the compilers with a common backend parallelizer and optimizer, enabling each language to realize important benefits from these performance enhancements (Figure 2-2). By providing language-specific front ends that translate source code to an intermediate representation, a

common, highly efficient global optimizer performs standard machine independent optimizations and loop parallelizations without regard for the language in use.

The nature of object-oriented programming and inherent structure of objects, classes, and methods, makes it difficult to automatically parallelize C++ source code. While Sun's C++ compiler is optimized, developers can make additional use of the threaded libraries supplied with the Solaris operating environment to further tune their applications.

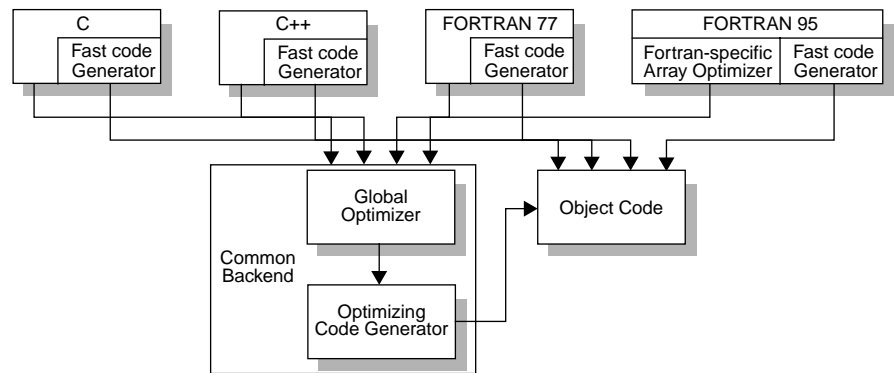


Figure 2-2 Forte Developer 6 compilers use a common backend for optimizing and parallelizing applications

Fortran and C: Auto Parallelizing and Pragma Assisted

All Sun optimizing compilers include capabilities to exploit instruction-level parallelism. These capabilities are further enhanced in Sun's C and Fortran multiprocessing compilers through sophisticated techniques, including loop parallelization, automatic user code restructuring, and directive and pragma support.

Loop Parallelization

The ability to perform multiple tasks in parallel speeds program execution and improves performance. Loops tend to dominate application runtime, and are the natural focus of automatic parallelization. Sun's C and Fortran compilers analyze source code and determine which loops are safe to parallelize. Loops whose iterations are independent of one another — where iteration order and parallel

execution do not affect the correctness of the result — are run in parallel. For this to be true, no memory location may be modified by more than one loop iteration. The compilers perform extensive data dependence analysis to ensure two different iterations of a loop cannot interfere with one another. If the compiler determines no data dependence exists, and it is profitable to execute the loop in parallel, the compiler automatically generates code that may be executed by multiple parallel threads.

To enable automatic parallelization and take advantage of multiple processors at runtime, the developer must use an automatic parallelization option such as `-xparallel` or `-xautopar`. Before application execution, the developer must set the `PARALLEL` environment variable, indicating the number of processors available to the application (Figure 2-3). If the target machine contains multiple processors, each thread generated will map to an independent processor, thereby speeding overall performance. Developers should note that failing to set the `PARALLEL` environment variable to be greater than one will result in no parallelization.

```
% cc -fast -xparallel example.c -lmopt -o example
%
% setenv PARALLEL 4
%
% example
```

```
% f90 -fast -xparallel example.f -o example
%
% setenv PARALLEL 4
%
% example
```

Figure 2-3 The Forte Developer C and Fortran compilers automatically parallelize loops to speed application performance

Some languages, such as C, use constructs which make it difficult for the optimizer to make aggressive automatic parallelization optimizations. Memory operations, for example, can benefit from optimizations when items do not overlap in memory. When they do, or if it appears that they might overlap, conservative optimizations must be performed to ensure accuracy and correct results. The Forte Developer C compiler includes the `-xrestrict` option, directing the compiler to assume that vectors, arrays, and pointers do not overlap. Without this assumption, virtually no C application can be parallelized. Developers should note that `-xrestrict` is best used with optimization level `-xO3` or higher.

More information on `-xrestrict` can be found in the *C User's Guide* located at <http://docs.sun.com> on the World Wide Web.

Automatic Loop Transformations

Sun's C and Fortran compilers perform extensive automatic restructuring of source code, exposing higher degrees of loop level parallelization. Several transformation techniques are employed, including loop distribution, loop fusion, and loop interchange.

- *Loop distribution*

Sometimes an application loop contains many statements that can execute in parallel and a few statements that cannot. Loop distribution separates the two types of statements. Sequential statements are organized into one loop, and those that can be parallelized are placed in a separate loop. Sun's C and Fortran compilers automatically perform extensive analysis to determine the safety and profitability of loop distribution.

- *Loop fusion*

When minimal work is performed by a parallel loop, the relative overhead of start up cost is high, lessening the performance benefit obtained from the parallelization effort. Loop fusion is used by the compiler to combine several small, adjacent loops into a single parallel loop, reducing execution overhead.

- *Loop interchange*

Nested loops provide an added challenge to the compiler. While it is generally more profitable to parallelize the outermost loop, it may not be safe to do so due to data dependencies. By interchanging the loops, however, the compiler is able to produce a loop that can be parallelized with significantly less overhead.

To enable full compiler-assisted parallelization, the developer must use the `-xparallel` option of the compiler (Figure 2-4). This enables the compiler's dependency analysis, loop distribution, loop fusion, and loop interchange mechanisms.

<pre>% cc -fast -xparallel example.c -o example % % setenv PARALLEL 4 % % example</pre>	<pre>% f90 -fast -xparallel example.f -o example % % setenv PARALLEL 4 % % example</pre>
---	--

Figure 2-4 The Forte Developer C and Fortran compilers speed application performance by transforming loops into code that is more easily parallelized

- *Scalar and array privatization*

Many times, the compiler may be able to parallelize a loop which has a data dependency. To create opportunities for parallel execution, the compiler utilizes multiple copies of a scalar or array variable, eliminating interference between loop iterations and enabling multiple processors to work on the task.

- *Reduction variables*

Cases arise when a real dependence between iterations of a loop and the variables causing the dependence cannot be made private. Using reduction variables, the compiler can assign a private variable to each thread. Before execution finishes, each thread's result is combined to obtain the final result.

Multiprocessing Language Extensions

Frequently there is insufficient information available to the compiler to enable it to make a good decision on the legality or profitability of parallelization. In these circumstances, developers can utilize parallelization directives, called pragmas, to tell the compiler to parallelize (or not to parallelize) the loop that follows the directive (Figure 2-5).

```
#pragma MP serial_loop_nested
for (i=0; i<100; i++) {
  #pragma MP taskloop
  for (j=0; j<1000; j++) {
    #pragma MP serial_loop
    for (k=0, k<100, k++) {
      ...
    }
  }
}
```

Figure 2-5 Forte Developer compilers support pragmas to enable the developer to force or prevent parallelization

Forte Developer 6 supports several parallel directives:

- TASKCOMMON, declares a COMMON block private
- DOALL, parallelizes the next loop
- DOSERIAL, does not parallelize the next loop
- DOSERIAL*, does not parallelize the next nest of loops
- INLINE, suggests the compiler inline the routines listed
- PIPELOOP, pipelines a loop

In addition, the `REDUCTION` directive has been modified to permit arrays to appear in the variables list.

Cray Parallelization Directives

Forte Developer 6 now supports Cray-style parallelization directives. Because the Forte compilers default to Sun-style directives, developers must compile with the `-mp=cray` option to utilize Cray-style parallelization directives. Developers should note that Cray-style directives requires explicit scoping of every scalar and array in a loop as either `SHARED` or `PRIVATE`, and that the mixing of unit compiled with both Sun and Cray directives can produce different results.

TASKCOMMON Pragma

Frequently developers are faced with the problem of needing to adapt serial programs to take full advantage of multiprocessor systems without significant source code modifications. Toward this end, developers need a way to employ *private* common blocks, a mechanism that facilitates the porting of traditional supercomputer applications to Solaris-based systems.

The `TASKCOMMON` directive in Forte Developer 6 declares variables in a global `COMMON` block as private. Every variable declared in a task common block becomes a private variable. Only named `COMMON` blocks can be declared `TASKCOMMON`. The `TASKCOMMON` directive must appear immediately after the defining `COMMON` declaration, and is effective only when compiled with `-xexplicitpar` or `-xparallel`. Otherwise, the directive is ignored and the block is treated as a regular common block.

Variables declared in `TASKCOMMON` blocks are treated as private variables in all the `DOALL` loops they appear in explicitly, and in the routines called from a loop where the specified common block is in its scope.

Runtime Checking of Common Block Inconsistencies

Declaring common blocks inconsistently can lead to incorrect results. This is especially true for `TASKCOMMON` blocks. The `-xcommonchk` option in Forte Developer 6 enables runtime checking of common block inconsistencies. It provides a debug check for common block inconsistencies in programs using `TASKCOMMON` and parallelization. If a common block declared in one source

program as a regular common block appears elsewhere on a `TASKCOMMON` directive, the program will stop with an error message indicating the first such inconsistency.

To enable runtime checking, developers must compile applications with `-xcommonchk=yes`; the default is `-xcommonchk=no`. Developers should use this option only during program development and debugging as its use can degrade application performance.

More information on pragmas can be found in the *Chapter 4, Parallelizing Sun ANSI/ISO C Code*, of the *Forte Developer C User's Guide*.

Sun Performance Library

Some of the most widely used libraries in existence today are the mathematical collections known as BLAS, LINPACK, LAPACK, FFTPACK and VFFTPACK. While these libraries provide a host of functions, they are not typically highly optimized or parallelized for a particular system. The Sun Performance Library™ contains highly optimized versions (in both scalar and parallel settings) of these libraries. With native interfaces to FORTRAN 77 and C, and accessibility from Fortran 90, the Sun Performance Library can boost application speed by a factor of four or more.

Developers of computationally intensive applications, such as those performing computational linear algebra and Fourier transforms, should use the Sun Performance Library. Although some applications may need to be modified to attain peak performance, many can benefit significantly from the use of the library without source code modifications or recompilations.

Specifically, application developers should consider the following common techniques for using the Sun Performance Library to improve performance:

- *Switch to using the Sun Performance Library*
Many applications are built using one or more of the seven libraries that comprise the Sun Performance Library. Because the Sun Performance Library maintains the same interfaces and functionality as these libraries, simply switching to Sun's optimized library can dramatically increase application performance without source code modification (Figure 2-6).

- *Use the Sun Performance Library to speed up other libraries*
Users of other mathematical libraries can substitute the BLAS included in their library with the one included in the Sun Performance Library. This is a helpful approach when an application has a dependency on proprietary interfaces in another library that prohibit it from being entirely replaced.
- *Use tools to restructure source code*
Other libraries may utilize different naming conventions than the subroutines in the Sun Performance Library. For many of these libraries, tools such as Pacific-Sierra's VAST/parallel, NA Software's LOFT90, and KAI's KAP can transform an application to use the Sun Performance Library without developer intervention.

The Sun Performance Library uses two modes of parallelization that are selectable at link time. The *dedicated mode* of parallelization delivers peak performance to applications using both the automatic parallelization features in Forte compilers and running on an MP machine that is dedicated to a single CPU-intensive application. The *shared mode* of parallelization should be used with applications that do not use automatic parallelization techniques (applications that make use of `libthread` for parallelization) or that run on a machine that is shared with other applications. Developers should choose the parallelization mode that best suits their application runtime environment.

```
% cc -fast -xparallel prg.c -xlic_lib=sunperf -o prg
%
% prg
```

```
% f90 -o prg -fast -xparallel prg.f -xlic_lib=sunperf -o prg
%
% prg
```

Figure 2-6 Use of the Sun Performance Library can significantly improve application performance

More information on the Sun Performance Library can be found in *Using Sun Performance Library*, listed in the references section at the end of this document, or at <http://www.sun.com/forte/fortran/documentation/index.html> on the World Wide Web.

Math Libraries

The `libm` math library bundled with the Solaris operating system contains the functions required by the various standards to which the Solaris operating system conforms. Developers focused on compute-intensive application performance will want to use the optimized math libraries and options provided with Forte Developer compilers for increased performance.

- *Optimized math library — `libmopt`*

Optimized versions of some of the routines in `libm` are provided in the library `libmopt`. The `libmopt` versions are generally noticeably faster. Unlike the `libm` versions, which can be configured to provide any of ANSI/POSIX, SVID, X/Open, or IEEE-style treatment of exception cases, the `libmopt` routines only support IEEE-style handling of these cases. The `libmopt` library may be linked with a C program by specifying the `-lmopt` flag at link time. It will be included automatically at link time for Fortran programs compiled with the `-fast` option. Care should be taken to ensure the `-lmopt` option precedes the `-lm` option during linking if `-lm` is also specified.

```
% cc -fast -xO5 -xparallel prg.c -lmopt -lm -o prg      % f90 -fast -xO5 -xparallel prg.f -o prg
%                                                     %
% prg                                                 % prg
```

Figure 2-7 C developers need to link applications with the `-lmopt` flag to take advantage of optimized math routines; Fortran compilation with `-fast` implicitly performs this linking

- *Vector math library — `libmvec`*

On SPARC systems running the Solaris 2.6, 7, or 8 operating environments, the library `libmvec` provides special routines that evaluate common mathematical functions for an entire vector of arguments. Parallel versions of the vector functions for multiprocessors are available in the `libmvec_mt.a` library. To use `libmvec_mt.a` or the single processor version `libmvec.a`, developers must link their applications with the `-lmvec` option. Developers should note that `libmvec_mt.a` will be linked automatically if `-xparallel` and `-lmvec` are specified.

```
% cc -fast -xparallel prg.c -lm -lmvec -o prg      % f90 -fast -xparallel prg.f -lm -lmvec -o prg
%                                                     %
% prg                                                 % prg
```

Figure 2-8 Developers need to link applications with the `-lmvec` option to include the vector math library

More information on `libmvec` can be found in the `libmvec(1)` man page.

- *Replace math library calls with vector versions of math routines — `xvector`*

The overhead associated with calling a routine multiple times often leads to reduced performance. The Forte Developer 6 `-xvector` option enables the compiler to replace math library calls in loops with single calls to faster vector versions of the math routines, where possible. When used, `-xvector` can significantly increase the performance of applications containing loops with large loop counts. Developers should note that the compiler will automatically notify the linker to include the `libmvec` and `libc` libraries in the load step if `-xvector` appears. However, to compile and link in separate steps, developers must specify `-xvector` at link time as well to correctly select these libraries. In addition, developers should be aware that the `-xvector` option also triggers the `-depend` option. Developers can include both `-xvector` and `-nodepend` at compile time to avoid this dependency analysis.

- *Use value-added mathematical functions — `libsunmath`*

The `libsunmath` library provided with the Forte Developer compilers contains a set of value-added mathematical functions, including:

- *Elementary transcendental and trigonometric functions in float and quad precision*
- *Financial functions*
- *Integral rounding functions*
- *IEEE standard recommended functions*
- *Functions that supply useful IEEE values*
- *Additive, linear congruential, and multiply-with-carry random number generators*
- *Random number shufflers*
- *Data conversion functions*
- *Functions that control rounding, mode, and floating-point exception flags*
- *Floating-point trap handling functions*

To take advantage of these routines, developers must link applications with the `-lsunmath` option. More information on the `libsunmath` library can be found in the *Numerical Computation Guide* located at <http://docs.sun.com> on the World Wide Web.

Developers should note that with Forte Developer 6 all math libraries are available in 64-bit versions. Unless otherwise noted, additional options do not need to be specified to link with these versions of the libraries; all Forte compilers will automatically select the correct set of libraries based on the `-xarch` option specified on the link line. More information on the math libraries can be found in the README file in the `<install_path>/SUNWspro/READMEs/` directory.

Controlling the Floating-Point Environment

Sun's floating-point environment implements the arithmetic model specified by the IEEE Standard 754 for Binary Floating Point Arithmetic. IEEE 754 arithmetic differs from older models in rounding and handling exceptions related to rounding, underflow, overflow, division by zero, and invalid operations such as zero divided by zero (0/0) and the square root of negative one (sqrt(-1)). By supporting user handling of exceptions, rounding, and precision, interval arithmetic and diagnosis of anomalies are possible.

IEEE arithmetic offers users greater control over computation than does any other kind of floating-point arithmetic, and simplifies the task of writing numerically sophisticated, portable programs. By supporting IEEE 754, Sun's floating-point environment enables the development of robust, high performance, portable numerical applications, and supplies the tools needed to investigate unusual program behavior. Forte Developer 6 provides several compiler options that enable developers to exploit or avoid features of IEEE 754 arithmetic:

- *Enable non-standard floating-point mode — fns*

On most SPARC systems it is possible to specify a non-standard floating-point mode that disables *gradual underflow*. This has the effect of causing tiny results to be flushed to zero rather than producing subnormal numbers, as well as causing subnormal operands to be silently replaced by zero. These SPARC systems do not support completely gradual underflow and subnormal numbers in hardware. On these systems, developers can use the `-fns` compiler option to enable non-standard floating-point mode when the program begins execution, thereby significantly improving performance of some applications at the expense of adherence to the IEEE standard.

More information on subnormal numbers can be found in the *Numerical Computation Guide* and the *Fortran Programming Guide* chapter on floating-point arithmetic for more information on subnormal numbers and related compiler options.

- *Enable trapping for floating-point exceptions — ftrap*

Because most exceptions are not significant and signals can degrade performance, the Forte Developer compilers do not generate a signal to interrupt applications for a floating-point exception automatically. To enable floating-point exception trapping, developers can employ the `-ftrap` option at compile time to set the IEEE 754 trapping modes that are established at

program initialization. Information on the type of conditions that can be trapped can be found in the *Fortran Programming Guide* available at <http://docs.sun.com> on the World Wide Web.

- *Initialize floating-point hardware to non-standard preferences — fnonstd*

The `-fnonstd` option enables hardware traps for floating-point overflow, division by zero, and invalid operation exceptions. These are converted into SIGFPE signals, and if the program has no SIGFPE handler, it terminates with a dump of memory (UNIX core dump). When this option is utilized the floating-point hardware is initialized to abort (trap) on floating-point exceptions, and flush underflow results to zero if the hardware so permits, rather than produce a subnormal number. Developers should note that `-fnonstd` is a macro for `-fns -ftrap=common`.

- *Set the IEEE rounding mode in effect at startup — fround*

The `-fround` option sets the IEEE 754 rounding mode that can be used by the compiler in evaluating constant expressions, and is established at runtime during the program initialization. When `-fround=tozero`, `-fround=negative`, or `-fround=positive`, the option sets the rounding direction to round-to-zero, round-to-negative-infinity, or round-to-positive-infinity, respectively, when the program begins execution. When `-fround` is not specified, `-fround=nearest` is used as the default and the rounding direction is round-to-nearest. Developers should note that to be effective, the main program should be compiled with this option.

- *Select floating-point optimization preferences — fsimple*

The `-fsimple` compiler option allows the compiler's optimizer to make simplifying assumptions concerning floating-point arithmetic. Several options are available:

- `-fsimple=0`, preserving strict IEEE 754 conformance and permitting no simplifying assumptions.
- `-fsimple=1`, not strictly conforming to IEEE 754, permitting conservative simplifications that result in numeric results of most programs remaining unchanged. With `-fsimple=1`, the optimizer can assume that: IEEE 754 default rounding/trapping modes do not change after process initialization; computations producing no visible result other than potential floating point exceptions may be deleted; computations with Infinity or NaNs ("Not a Number") as operands need not propagate NaNs to their results; and computations do not depend on the sign of zero. However, in this mode the optimizer is not allowed to optimize completely

without regard to roundoff or exceptions. In particular, a floating-point computation cannot be replaced by one that produces different results with rounding modes held constant at run time.

- `-fsimple=2`, permitting aggressive floating point optimizations that may cause many programs to produce different numeric results due to changes in rounding. For example, `-fsimple=2` permits the optimizer to attempt to replace repeated computations of x/y with $x*z$, where $z=1/y$ is computed once and saved in a temporary variable, eliminating the costly divide operation. Even with `-fsimple=2`, the optimizer still is not permitted to introduce a floating point exception in a program that otherwise produces none.

Developers should note that all units of a program should be compiled with the same `-fsimple` option to ensure consistent results.

- *Detecting floating-point overflow conditions — `fpovery` (Fortran only)*

The `-fpovery=yes` option forces the compiler to check for and detect floating-point overflow conditions in formatted output. Developers should specify the `-fpovery=yes` option to ensure the I/O library will detect runtime floating-point overflows in formatted input and return an error condition (1031) if a problem arises. The default is no such overflow detection (`-fpovery=no`). Developers should note that `-fpovery` is equivalent to `-fpovery=yes`.

- *Force IEEE 754 style return values — `xlibmieee` (C only)*

The `-xlibmieee` option forces the compiler to return IEEE 754 style values for math routines in exceptional cases. In such cases, no exception message is printed. Developers should note that when using `-xlibmieee`, the variable `errno` should not be relied upon for error information.

- *Force unsuffixed floating-point constants to be single precision — `xsfpconst` (C only)*

The `-xsfpconst` option forces the compiler to represent unsuffixed floating-point constants as single-precision numbers, rather than the default mode of double-precision. Developers should note that `-xsfpconst` cannot be used with the `-xc` option.

mediaLib™ and the VIS™ Instruction Set

Specialized instructions that execute complex multimedia and networking operations can dramatically increase a processor's throughput and hence application performance. The VIS Instruction Set features of Sun UltraSPARC processors enable the execution of complex multimedia and networking

operations that typically require dozens of clocks to be performed in a single clock cycle. VIS instructions include support for image processing, video compression, pixel format and conversion, data transfers, and animation speed up. The use of these accelerated functions enables developers to build high performance multimedia applications with ease.

The `mediaLib` library is a collection of C functions that support multimedia operations and the VIS Instruction Set. When executed on UltraSPARC platforms, these library routines take full advantage of the on-board capabilities of the UltraSPARC processor, with execution speed improved by as much as seven times. To utilize `mediaLib`, developers must compile their applications to include `/opt/SUNWmlib/lib` (Figure 2-9). A version of `mediaLib` is available for each SPARC architecture (Version 7, 8, 8+, 9) and can be downloaded from <http://www.sun.com/sparc/vis>.

```
% cc -fast -xparallel -xarch=ultra -xdepend prog.c -I/opt/SUNWlib/include -L /opt/SUNWmlib/lib -lmlib -o prog
```

Figure 2-9 The use of `mediaLib` can significantly improve application performance

More information on `mediaLib` and the VIS Instruction Set can be found at <http://www.sun.com/sparc/vis> and in the references listed at the end of this document.

Optimization Levels

Choice of the proper compiler options is a key step in improving performance. Sun compilers offer a wide range of options that affect object code and application performance.

No compiler optimizations are performed by the compilers unless a `-xO` option is specified explicitly or implicitly. In nearly all cases, specifying an optimization level for compilation improves program execution performance. On the other hand, higher levels of optimization increase compilation time and may significantly increase code size. Forte compilers enable optimizations to be turned on or off to suit developer preferences.

- For most cases, level `-xO3` is a good balance between performance gain, code size, and compilation time.
- Level `-xO4` adds automatic inlining of calls to routines contained in the same source file as the caller routine, among other things. Developers can also specify automatic inlining of routines via the `-xinline=auto` option, or inhibit the inlining of a function by setting `-xinline=%no function-name`.

- Level `-xO5` aggressively performs transformations, and should be used when performance is the paramount concern. Level `-xO5` should only be used with profile feedback. If the `-xprofile` option is not specified, `-fast` defaults to using the `-xO4` optimization level.

In general, levels above `-xO3` should be specified only to those routines that make up the most compute-intensive parts of the program and thereby have a high certainty of improving performance. To identify these areas, developers can use the profiling tools provided with Forte Developer compilers and debuggers.

Adjusting the Maximum Optimization Level

Sometimes high levels of optimization can lead to unexpected results. While it is often straightforward to locate the problematic file, it can be difficult to zero in on the errant function. Working in conjunction, the `-xmaxopt` compiler option and the `OPT=` pragma available in Forte Developer 6 give developers a mechanism to help them control optimization on a per function basis.

The `-xmaxopt` compiler option enables optimization pragmas and sets the maximum optimization level to be used for compiling an application. This option enables the `C$PRAGMA SUN OPT=n` Fortran directive, or the `#pragma optlevel (func..)` C directive, when it appears in the source file; without it, the compiler treats such lines as comments. If a pragma appears with an optimization level greater than the maximum level on the `-xmaxopt` flag, the compiler uses `-xmaxopt` level. Optimization levels `-O1` through `-O5` are available, with level `-O5` being the default.

More information on `-xmaxopt` and `OPT=` can be found in the Fortran and C manuals at <http://docs.sun.com> on the World Wide Web.

Optimizing for Speed — The -fast Option

Developers are constantly trying to find the right balance of options that maximizes application performance. To help this effort, the Forte Developer compilers provide a grouping of select options that optimize for speed of execution without excessive compilation time. This `-fast` option provides close to maximum performance for many applications. The `-fast` option is a macro that includes the individual compiler option components identified in Figure 2-10.

```
-xtarget=native -xO5 -dalign -fns -fsimple=1 -ftrap=%none -fsingle -libmil
```

Figure 2-10 The `-fast` option expands into multiple compiler option components

The individual components of `-fast` can be overridden by putting the proper individual flag after the `-fast` flag during compilation. For example, developers can override the `-fsimple=1` option and disable the `-xlibmopt` option by adding `-fsimple=2` and `-xnolibmopt` after the `-fast` option (Figure 2-11).



Figure 2-11 The individual components of the `-fast` option can be overridden

Developers should note that because the `-fast` option is defined as a particular selection of compiler options, it is subject to change from one release to another, as well as between compilers. In addition, some of the component options selected by `-fast` may not be available on some platforms. In addition, care needs to be taken if application compilation and linking are performed separately. Developers should ensure that applications are both compiled and linked with `-fast` to ensure proper behavior.

Developers should note that the `-fast` option no longer generates 32-bit generic binaries that can run on any SPARC processor with the release of Forte Developer 6. As a result, the use of the `-fast` option may prove inappropriate for generic production environments. Indeed, for maximum performance, developers may choose other performance enhancing options and compile for a specific processor architecture. For more information, see the discussion on the `-xarch` option later in this chapter.

More information on `-fast` and related options can be found in the programming and user guides for each compiler located at <http://www.docs.com>.

Profile Feedback

Optimization and performance tuning is an art that depends heavily on being able to determine what to optimize or tune. *Profile feedback* is one means of improving optimization by gathering information about the frequency of execution of various parts of a program. Profile feedback has two phases, collect and use. During the collect phase, the source code is instrumented to gather data — counters are inserted into the source code to facilitate determining the number of times the code was executed. During the use stage, the compiler uses the information it gathered in the collect stage to optimize the application.

The compiler applies its optimization strategies at level `-xO3` and above much more efficiently if combined with the `-xprofile=use` option. With this option, the optimizer is directed by a runtime execution profile produced by the program (compiled with `-xprofile=collect`) with typical input data.

The feedback profile indicates to the compiler where optimization will have the greatest effect. This is particularly important with optimization level `-xO5`. Indeed, levels `-xO3` and `-xO4` typically produce faster code, while level `-xO5` generally does not do so unless profile feedback is also used. In fact, the use of `-xO5` without profile feedback can produce inferior results than the use of the `-xO4` optimization level alone. Developers should note that if the `-xprofile` option is not specified, `-fast` defaults to using the `-xO4` optimization level.

Figure 2-12 identifies how developers can achieve profile collection with higher optimization levels. The first compilation generates an executable that produces statement coverage statistics when run. The second compilation uses this performance data to guide the optimization of the program.

```
% cc -fast -xO5 -xprofile=collect prg.c      % f90 -o prg -fast -xO5 -xprofile=collect prg.f
%                                           %
% prg                                       % prg
%                                           %
% cc -fast -xO5 -xprofile=use prg.c        % f90 -o prg -fast -xO5 -xprofile=use prg.f
%                                           %
% prg                                       % prg
```

Figure 2-12 Forte Developer compilers include profile feedback options that direct compiler optimizations

Performance Tools

Forte Developer compilers include a variety of tools to analyze source code, aid problem isolation, and provide developers with the information they need to finely tune their applications for maximum performance.

Forte Developer Performance Analyzer

The art of performance tuning is rarely seen as a debugging exercise. Indeed, the debugging of applications is often solely characterized by the finding and fixing of problems related to program accuracy. As computer systems continue to become more powerful, application performance is emerging as a critical factor, with bad performance increasingly considered a program failure. Developers are now keenly aware that they must streamline critical sections of source code as well as locate programmatic errors and coding deficiencies without impacting application accuracy.

Once an application has been designed, developed and debugged, developers can turn their attention to making it perform well. Forte Developer 6 includes two powerful applications that work together in the debugging process to locate issues that affect program performance — the Sampling Collector and the Sampling Analyzer (Figure 2-13).

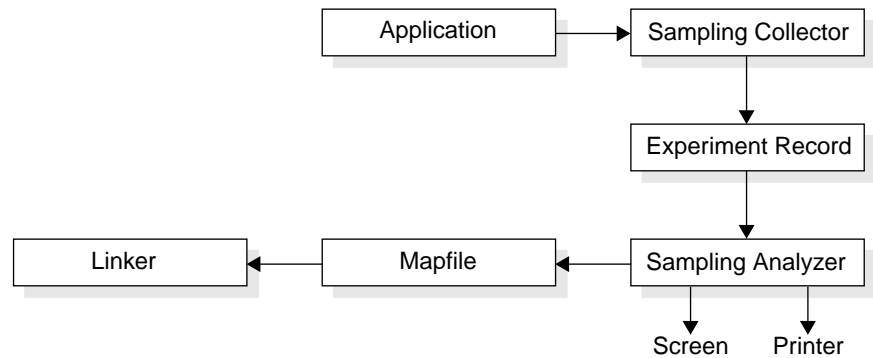


Figure 2-13 The Sampling Collector and Sampling Analyzer work together in the debugging process to locate issues that affect program performance

Programs to be analyzed may be compiled with any level of parallelization and optimization. To see source code, developers must compile applications with the `-g` option. It is important to note that the `-g` option no longer inhibits parallelization and optimization, and enables performance measurement to be performed on applications that are compiled for production use.

Sampling Collector

The Sampling Collector gathers performance data during application execution, saving it to an experiment file to be used later during the analysis process. As the program runs through the Debugging Window, performance data is collected (Figure 2-14). The Sampling Collector enables developers to obtain information on:

- Clock-based profiles
- Thread-synchronization delay events and wait time
- Operating system summary information
- Hardware-counter overflow profiles, on systems where the hardware supports it (UltraSPARC-III and later)
- Global information, including execution statistics and address-space data

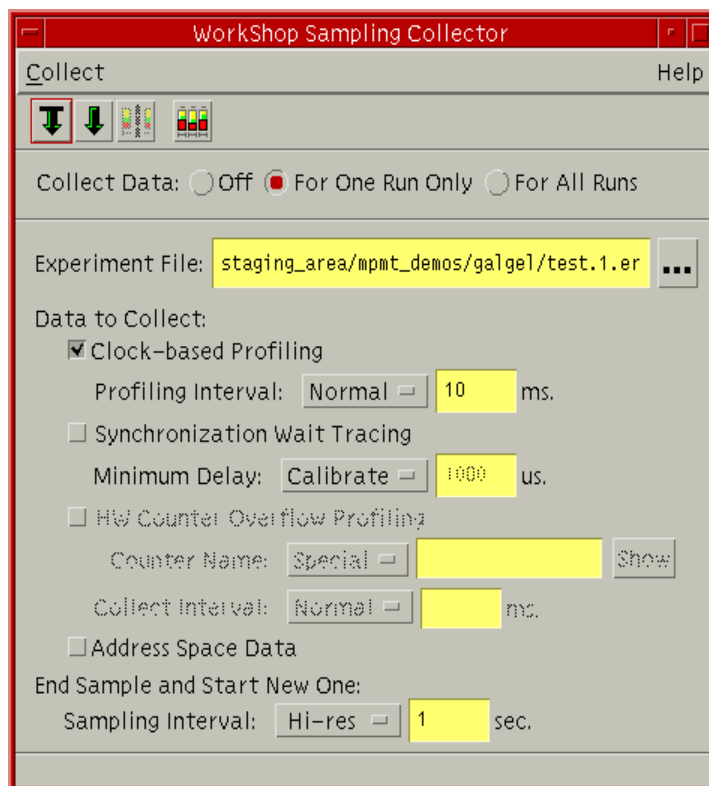


Figure 2-14 The Sampling Collector gathers performance data about a program

Collect Command

Data can also be collected using the `collect` command. The `collect` command gathers the same performance data that can be obtained using the Sampling Collector. All data can be converted into performance metrics computed against functions, callers and callees of any function, and against source and disassembly representations of the target program. It also records global data with periodic or manual sampling.

It is important to note that programs intended to be targets for the collector may be compiled with any level of optimization, but must use dynamic linking. Statically linked programs will result in a debugger error when an attempt is made to run the program for data collection. Developers should note that the `collect` command does not require the use of the `dbx` utility.

Sampling Analyzer

After data is collected with the Sampling Collector or the `collect` command, it can be viewed through the Sampling Analyzer component of Forte Developer. The Sampling Analyzer measures and graphically displays an application's performance profile, suggesting ways to improve performance. Developers can collect a variety of performance data types, and can control the data collection process while an application is running. By focusing on the areas where performance problems occur, developers can test hypotheses about a program's behavior.

When the analyzer is invoked on an experiment, it reads the experiment data and presents a *function list* with metrics. Metrics may be either exclusive or inclusive. Exclusive metrics represent usage within the function, while inclusive metrics represent usage within the function and all the functions it called. Developers can analyze the collected data and obtain information in several critical areas. Initially, a default set of metrics based on the data collected is shown. Time metrics are shown as seconds, presented to millisecond precision. Percentages are shown to a precision of 0.1 %.

- *Clock-based profiling data*, a summary of process state times, including user CPU, total lightweight process time, wall-clock time, system CPU time, system wait time, text page fault time, and data page fault time
- *Thread-synchronization wait tracing*, the time spent waiting on calls to thread-synchronization routines in the threads library
- *Synchronization wait tracing*, the number of events recorded and the number of seconds over threshold waiting on calls to thread-synchronization routines
- *Hardware-counter overflow profiling*, information on the threads and lightweight processes running during CPU hardware counter overflows, including data on instruction-cache misses, data-cache misses, cycles, and instructions issued or executed
- *Address space*, the reference behavior of both text pages and data pages
- *Execution statistics*, overall statistics on the execution of the application

More information on available metrics can be found in the `collector(1)` man page.

Viewing Performance Data

To aid application analysis, the Collecting Analyzer provides several ways for developers to view collected performance data, including data display at the function or load object level. Developers can control which metrics are shown, as well as the order in which they appear.

- *Function list*

The Function List provides information about process times during part or all of program execution, including the average time spent in various process states (Figure 2-15). Additional views provide information on sample details, address space, pages and segments, and execution statistics.

Excl. User CPU sec.	Incl. User CPU sec.	Name
73,990	73,990	<Total>
11,830	11,830	gpf_work
10,710	10,720	cputime
9,270	9,270	sigtime_handler
7,800	7,810	underflow
6,690	6,700	icputime
6,430	6,430	real_recurse
6,090	6,120	muldiv
2,460	2,460	bounce_a
1,340	1,340	inc_brace
1,320	1,320	inc_body
1,320	1,320	inc_func
1,230	1,230	gethrtime
0,960	0,960	gettimeofday
0,930	0,930	inline_code
0,910	0,910	s_inline_code
0,870	0,870	ext_inline_code
0,640	0,640	tailcall_c
0,620	0,620	gethrtime
0,610	1,250	tailcall_b
0,600	1,850	tailcall_a
0,460	0,460	macro_code
0,460	3,510	sustime

Figure 2-15 The Sampling Analyzer allows developers to view data in multiple ways

- *Caller-Callees screen*

The Callers-Callees displays a selected function in the center of the screen, with callers of that function in a panel above, and callees of that function in a panel below (Figure 2-16).

In addition to showing the metric values, *attributed* metrics are also displayed for each function. For the center function, the attributed metric represents the exclusive metric for that function, while for the callees below, it represents the proportion of that callee's metric that is attributable to calls from the center function. The sum of attributed metrics for the callees and the center function add up to the inclusive metric for the center function. For the callers above, the attributed metrics represent the inclusive metric value from the center function, as attributed up the callstack to its callers. The sum of the attributed metrics for all callers adds up to the inclusive metric for the center function. Selecting a different function in the function list, or an entry in the current caller or callee list updates the Callers-Callees window to center it on the selected function.

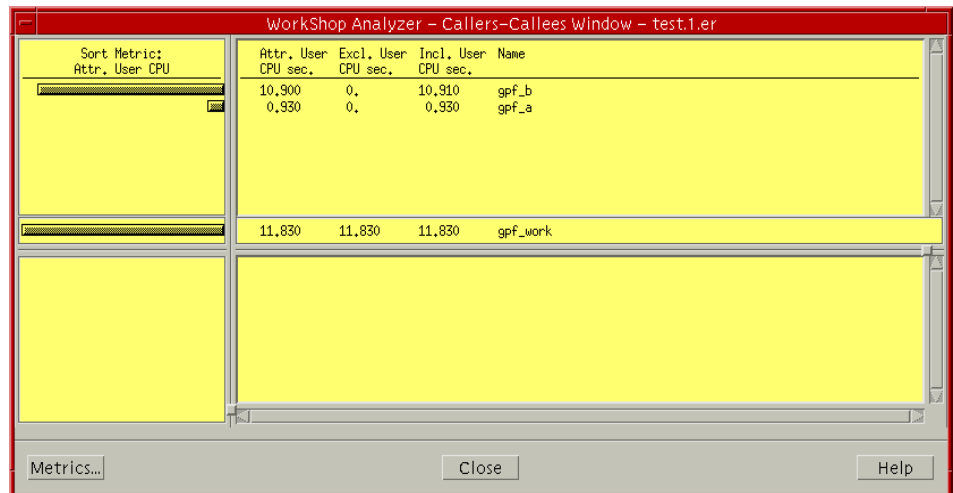


Figure 2-16 The Caller-Callees screen displays metric values and attributed metrics for functions

- *Annotated source*

The source of a selected function can be viewed with annotations of performance metrics for each source line, and any compiler commentary shown interleaved with the source to which it pertains (Figure 2-17). In the rare case where the same source file is used to compile more than one object file, the source display shows the performance data for the object file containing the selected function.

```

52      45. ! del = (iend - istart) / 1000000000.
53      46. ! vdel = (ivend - ivstart) / 1000000000.
54      47. ! write(10, 9000) del, vdel, 'asub_alone'
55      48. ! del = (jend - jstart) / 1000000000.
56      49. ! vdel = (jvend - jvstart) / 1000000000.
57      50. ! write(10, 9000) del, vdel, 'asub1_alone'
58      51.
59      52. del = (jend - istart) / 1000000000.
60      53. vdel = (jvend - ivstart) / 1000000000.
61      54. write(10, 9000) del, vdel, 'asub_'
62      55.
63      56. ! Do copyin test
64      57. istart = gethrtime()
65      58. ivstart = gethrtime()
66
67      In the call below, parameter number 1 caused a copy-in and a copy-out -- Loops inserted
68      Loop below is parallelized: no hint available
69      Loop below is parallelized: no hint available
70      Loop below is parallelized: loop may or may not hold enough work to be profitably parallelized
71      Loop below is parallelized: loop may or may not hold enough work to be profitably parallelized
72      59. call copytest(a(100:1:-2, 36:79), 44 )
73      60. iend = gethrtime()
74      61. ivend = gethrtime()
75      62. del = (iend - istart) / 1000000000.
76      63. vdel = (ivend - ivstart) / 1000000000.
77      64. write(10, 9000) del, vdel, 'copytest_'
78      65.
79      66. ! Do contained subroutine test
80      67. istart = gethrtime()
81      68. ivstart = gethrtime()
82      69. call wrapper(rtr)
83      70. iend = gethrtime()
84      71. ivend = gethrtime()
85      72. del = (iend - istart) / 1000000000.

```

Figure 2-17 The Annotated Source screen

- *Annotated disassembly*

Annotated disassembly gives developers an assembly code listing of the instructions of a function or module, along with the performance metrics associated with each instruction. Annotated disassembly can be displayed in several ways. Each instruction is annotated with a source line number as reported by the compiler, its relative address, the hexadecimal representation of the instruction, the assembler ASCII representation of the instruction. Wherever possible, call addresses are resolved to symbols.

More information on annotated disassembly can be found in the *Analyzing Program Performance with Sun WorkShop* located at <http://docs.sun.com> on the World Wide Web.

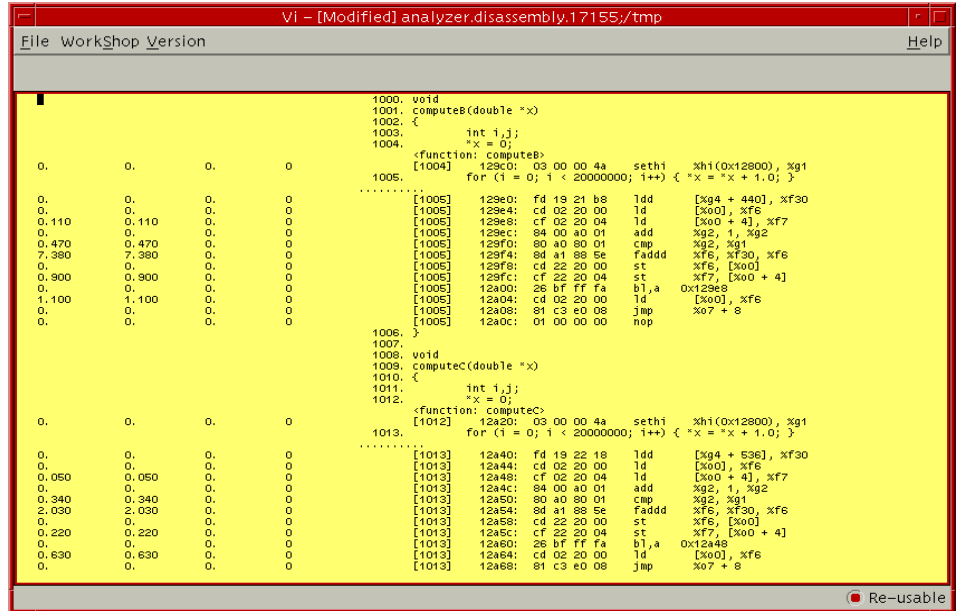


Figure 2-18 The Annotated Disassembly screen displays interleaved source and compiler commentary for selected functions

er_print

The `er_print` command processes experiments files generated using the Sampling Collector or the `dbx collector` command in exactly the same manner as the Sampling Analyzer. Because the `er_print` command does not have a graphical user interface, it generates ASCII output of the various displays supported by the Sampling Analyzer. All generated reports match the Sampling Analyzer `print` option in content.

More information on `er_print` can be found in the `er_print(1)` man page.

Mapfile Generation

To further help in rebuilding programs with improved performance, the analyzer can identify a potentially improved ordering for loading functions into the program's address space. Using the `create mapfile` option, developers can create a mapfile and then adjust their makefiles to use the order it specifies, resulting in

an executable with reduced working-set size. Developers must compile applications with the `-xF` option to enable the reordering of functions in the core image. Once compiled, developers can run the Analyzer to generate a map file that optimizes the ordering of the functions in memory, depending on how they are used together. Developers must then link the applications using the `-Mmapfile` option to build an executable file that can be directed to use that map. Each function from the executable file is placed in a separate section.

Developers should note that the reordering of subprograms in memory is useful only when the application text page fault time is consuming a large percentage of the application time. Otherwise, reordering may not improve overall application performance.

More information on the Sampling Collector and Sampling Analyzer can be found in the *Analyzing Program Performance with Sun WorkShop* guide available at <http://docs.sun.com> on the World Wide Web.

Call Graph Profiles and gprof

Performance analysis tools provide a variety of analysis levels. Analysis levels vary from simple timing of a command to a statement-by-statement analysis of a program. While a flat profile can provide valuable data for performance improvements, sometimes the data obtained is not sufficient to point out exactly where improvements can be made. A more detailed analysis can be obtained by using the *call graph profile* to display a list identifying which modules are called by other modules, and which modules call other modules.

To enable call graph generation, developers must compile and link an application for call graph profiling using the `-xpg` option. Once the program is compiled in this manner, call-graph profile data is sent to a file called `gmon.out` at the end of each run. The `gprof` command can then be used to interpret the results of the profile. For example, developers may identify how many times a function is called, or if it is heavily recursive (Figure 2-19). These analyses may point the developer to algorithms that require modification or imply the need for parallelization optimizations.

Developers should note that `gprof` assumes all calls to a function take the same amount of time when computing cumulative time. For many functions, this assumption is not true and can lead to misleading data. The Performance Analyzer does not make this assumption.

```

granularity: each sample hit covers 2 bytes(s) for 0.02% of 65.08 seconds
%
time    self  descendents  called+self  called/totalchildren  nameindex
[3]    97.8  0.00  63.66
        0.00  63.66
        0.98  14.59
        1.36  9.55
        1.25  9.51
        0.04  8.45
        5.11  1.71
        0.00  6.21
        ...
[12]   9.5  6.16  0.00
        9.5  6.16
        ...

%      cumulative  self  self  total
time  seconds      seconds  calls  ms/call  ms/call  name
14.2  23.75         9.40   200    47.00   47.00   cholsky_ [8]
14.2  33.14         9.39   400    23.48   23.48   vpenta_ [9]
9.3   39.30         6.16   100    61.60   61.60   mxm_ [13]
9.1   45.31         6.01   201    29.90   29.90   cfft2d1_ [14]
7.7   50.42         5.11    2     2555.02  3411.96  gmtry_ [11]
4.4   53.34         2.92   54950000.00  0.00   __log [17]

```

Figure 2-19 Example gprof output

Multiplatform Support — 32-bit and 64-bit Applications

With the price of memory continuing to tumble, organizations are reaping the performance advantages of keeping more application data in main memory rather than on disk. Although few require an entire 64-bit address space, the impact of more than 32 bits of address space benefits a wide variety of commercial and high-performance computing applications.

Developers can make the transition to 64-bit interfaces easily, and at their leisure. The Solaris environment delivers exactly the same application programming interfaces (APIs) to both 32-bit and 64-bit applications. The behavior of each interface is exactly the same in 32-bit and 64-bit modes because they use common code; only the initial system call processing differs. As a result, the transition to 64-bit processing for almost all applications requires only some code modifications to match the industry-standard 64-bit programming language data models, and a change to the `-xarch` compiler flag.

Nevertheless, developers need to understand the impact of architecture choice. Applications which are compiled to take advantage of UltraSPARC-specific instructions and the Visual Instruction Set (`-xarch=v8plusa`, `-xarch=v9a`, `-xarch=v8plusb`, `-xarch=v9b`) will potentially outperform those compiled for other architectures. However, applications compiled to take advantage of these special, high performance processor features are not binary compatible with older systems and those that do not employ the UltraSPARC processor architecture. Consequently, developers must make a conscious choice and evaluate the trade-offs between the possible performance gains from these enhancements and any resulting binary incompatibility.

Specifying Processors — `xarch`

Developers should also identify the type of processor in use, enabling the compiler to generate code optimized for that processor's instruction set. If this option is used with optimization, the appropriate choice can provide good performance of the executable on the specified architecture.

The Forte Developer compilers support a variety of SPARC architectures:

- **Generic** (`-xarch=generic`)
This option uses only the instructions common to all SPARC processors, ensuring the application binary runs on all Sun platforms.
- **Version 7** (`-xarch=v7`)
This option uses the best instruction set for good performance on the V7 architecture. Typical SPARC V7 systems include the SPARCstation™ 1 and SPARCstation 2.
- **Version 8** (`-xarch=v8`)
This option uses the best instruction set for good performance on the V8 architecture. A typical V8 system uses the SuperSPARC™ processor, such as the SPARCstation 10.
- **Version 8plus** (`-xarch=v8plus`)
This option utilizes the 32-bit subset of the Version 9 architecture. This compiler option utilizes the V9 architecture without the VIS instructions. Developers must be aware that V8plus binaries will not run on a V7 or V8 system. All UltraSPARC-based systems are also V8plus machines.
- **Version 8plusa** (`-xarch=v8plusa`)
This option uses the best instruction set for good performance on the UltraSPARC architecture, but limited to the 32-bit subset defined by the

V8plus specification. V8plusa incorporates the V8plus architecture plus UltraSPARC-specific instructions and the VIS Instruction Set. Developers must be aware that V8plusa binaries will not run on a V7 or V8 system. All UltraSPARC-based systems are also V8plusa machines.

- **Version 8plusb (-xarch=v8plusb)**
This option uses the best instruction set for good performance on the UltraSPARC-III processor, but limited to the 32-bit subset defined by the V8plus specification. Support for the VIS Instruction Set is included. Developers must be aware that V8plusb binaries will only run on UltraSPARC-III systems.
- **Version 9 (-xarch=v9)**
This option uses the 64-bit Version 9 instruction set. This is supported on UltraSPARC systems running the Solaris 7 or Solaris 8 operating environments.
- **Version 9a (-xarch=v9a)**
This option uses the 64-bit Version 9 instruction set and the Visual Instruction Set supported on UltraSPARC systems running the Solaris 7 or Solaris 8 operating environments.
- **Version 9b (-xarch=v9b)**
This option uses the 64-bit Version 9 instruction set and the Visual Instruction Set supported on UltraSPARC-III systems running the Solaris 8 operating environment. Developers should note that V9b binaries will only run on UltraSPARC-III systems.

UltraSPARC Compilers: Built for Speed

The performance of some programs may benefit if the compiler has an accurate description of the target computer hardware. When program performance is critical, the proper specification of the target hardware can be very important. This is especially true when running on newer, high performance SPARC processors, such as UltraSPARC. Two factors are critical in identifying hardware: the *target system* and *processor architecture*.

Identifying Target Systems — xtarget

To identify a target system, developers must specify a platform during compilation using the `-xtarget` option when compiling. For any given system name (for example, `ultra`, for an UltraSPARC-based machine), `-xtarget`

expands into a specific combination of `-xarch`, `-xcache`, and `-xchip` that properly matches that system. The optimizer uses these specifications to determine strategies to follow and instructions to generate.

When the execution system is not known, developers should compile for a generic architecture (`xtarget=ultra`), although this may produce suboptimal performance. Performance-critical applications should ensure that the `-xarch` option is used appropriately, as described in the previous section.

Specifying Code Address Space — `xcode`

In order for applications to be efficient, they need to take full advantage of the code address space of the machines on which they run. With Forte Developer 6, programmers can employ a more flexible compiler flag, `-xcode=value`, to specify the code address space of a binary object. With this compiler flag, 32-bit, 44-bit, or 64-bit absolute addresses can be generated, as well as small and large model position-independent code (Table 2-1).

Code Address Space Generation Values	
abs32	Generates 32-bit absolute addresses. Code+data+bss size is limited to 2^{32} bytes, the default on 32-bit platforms. This option can be used with the <code>-xarch=generic</code> , <code>v7</code> , <code>v8</code> , <code>v8plus</code> , <code>v8plusa</code> , <code>v8plusb</code> options.
abs44	Generates 44-bit absolute addresses. Code+data+bss size is limited to 2^{44} bytes, and is available only on 64-bit platforms. This option can be used with the <code>-xarch=v9</code> , <code>v9a</code> , <code>v9b</code> options.
abs64	Generates 64-bit absolute addresses. Code+data+bss size is limited to 2^{64} bytes, and is available only on 64-bit platforms. This option can be used with the <code>-xarch=v9</code> , <code>v9a</code> , <code>v9b</code> options.
pic13	Generates small model position-independent code. References to at most 2^{11} unique external symbols on 32-bit platforms, or 2^{10} on 64-bit platforms, are permitted. This is equivalent <code>-pic</code> .
pic32	Generates large model position-independent code. References to at most 2^{30} unique external symbols on 32-bit platforms, or 2^{29} on 64-bit platforms, are permitted. This is equivalent to <code>-PIC</code> .

Table 2-1 Developers should use the `-xcode` compiler option to specify code address space on SPARC platforms

Specifying Cache Properties — xcache

Developers can also specify cache properties, enabling the optimizer to take full advantage of the processor and system architecture. While the `-xcache` option is part of the `-xtarget` option, it can be used alone to override default values and ensure maximum performance optimizations. When specifying cache properties for the optimizer, developers can identify level 1, 2, and 3 cache values, or use a default cache scheme that provides good performance on all processors.

The size of the cache, size of a cache line, and associativity of the cache should all be defined for the optimizer if the `generic` option is not used. To specify cache properties, developers must use the `-xcache=xx/yy/zz` option, where `xx` is the size of the cache in KB, `yy` is the size of a cache line in bytes, and `zz` is the associativity of the cache. For example, `-xcache=16/32/4:1024/64/1` specifies a 16 KB level 1 cache with a 32 byte line size that is 4-way associative, and a level 2 cache that is 1024 KB with a 64 byte line size that is direct mapped.

Cache Blocking

While the `-xcache` option attempts to perform the best optimizations for a given architecture, it cannot take into account the intent of the application source code. Developers that wish to maximize the use the processor cache can employ *cache blocking* — a technique that increases the cache-hit rates of the program by increasing the reuse of the data present in the cache. By assuming that a data item is likely to be reused again soon, or that neighboring data is likely to be accessed next, cache blocking reduces cache misses, thereby increasing application performance.

Many loop computations, such as matrix multiplications, are vulnerable to cache misses, particularly when the size of the array exceeds the cache size. By employing cache blocking techniques, developers can restructure source code so that computations are performed on contiguous cache regions. If architected properly, overall memory traffic can be decreased significantly, with resident data being used for multiple calculations. This technique is most beneficial to computations with a high operation count to memory access ratio, such as those found in finite element analysis, combustion analysis, and simulations. For example, a matrix multiplication of $N \times N$ matrices requires N^3 operations and N^2 memory operations. This ratio indicates the possibility of N operations being performed for every memory access, enabling the potential for data reuse.

Figure 2-20 identifies three cases that exemplify the benefits that can be achieved using cache blocking. In each case, each UltraSPARC processor employed a 512 KB external cache. Performing a matrix multiplication on a 320x320 array yielded a 41 percent performance gain — and the benefits scaled to over 90 percent with a larger array and multiple processors.

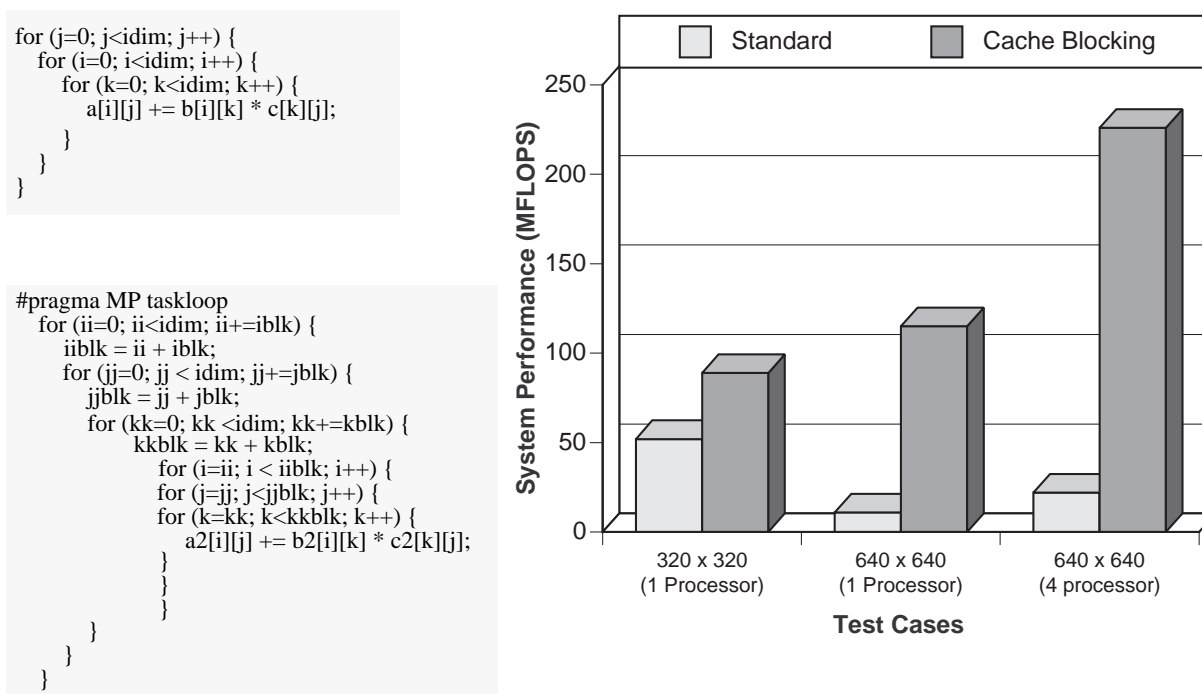


Figure 2-20 Cache blocking improves application performance

Insert Padding — `pad`

Most processors prefer finding and locating items on specific boundaries. Some programming languages, such as Fortran and C, can place items in locations different than the processor would prefer. The new `-pad` option in the Forte Fortran compiler enables developers to instruct the compiler to insert padding between arrays or character variables, or in common blocks. The extra padding positions the data to make better use of the cache. Developers must set `-pad=local` to add padding between adjacent local variables, and `-pad=common` to add padding between variables in common blocks.

More information on the `-pad` option can be found in the *Fortran Programming Guide* located at <http://docs.sun.com> on the World Wide Web.

xsafe=mem

Using this option allows the compiler to assume no memory-based traps occur. It grants permission to use the speculative load instruction on V9 machines, and is only effective if `-x05` and `-xarch=v8plus` (or `v8plusa`, `v8plusb`, `v9`, `v9a`, `v9b`) are also specified. This has the consequence of possibly changing application behavior. Programs that would ordinarily fail with a memory access violation are allowed to continue. While no memory access violations will be detected, logic errors may still exist, potentially affecting the developer's ability to detect certain programming errors. To take full advantage of this option, developers should use profile feedback as well.

xmemalign

Memory access errors can be a source of frustration for developers. The Forte Developer 6 compilers now support `-xmemalign`, enabling the compiler to control the code generated for potentially misaligned memory accesses, as well as control program behavior in the event of a misaligned access.

xprefetch

One of the ways to increase application performance is to ensure they take full advantage of the unique, high performance capabilities of the processors on which they run. Prefetch instructions, such as those supported in Sun's UltraSPARC-II and UltraSPARC-III processors, help all processor execution units to remain busy by fetching data and instructions before they are needed in the pipeline, thereby increasing overall performance. The new `-xprefetch` option supported by Forte Developer 6 enables the compiler to generate prefetch instructions where appropriate, potentially improving performance on UltraSPARC-II and UltraSPARC-III based systems. Developers should note that the `-xprefetch` option can be used when compiling with `-xarch=v8plus`, `-xarch=v8plusa`, `-xarch=v8plusb`, `-xarch=v9`, `-xarch=v9a`, `-xarch=v9b`, or any `-xtarget` platforms that include these architectures.

More information on Sun compiler technology, products, and options can be found at <http://www.sun.com/workshop> on the World Wide Web.

Sun's Guidelines for Achieving Maximum Application Performance

Developers will find that Forte Developer 6 offers a variety of tools and techniques that enable the construction of fast, efficient, and reliable code. Specific steps can be taken to achieve incremental benefits and the best application performance possible on both uniprocessor and multiprocessor systems.

Uniprocessor Environments

- *Use dynamic linking to system libraries*

By dynamically linking to system libraries, applications can automatically take advantage of the latest library versions tuned for the operating system and SPARC platform on which the application is running.

- *Compile for fast code*

The first step toward achieving better performance is to compile the application for fast code using the `-fast` option.

- *Compile for additional dependence analysis*

If additional performance is required, developers can compile the application with the `-xdepend` option. Developers should note that this option is included automatically with the `-fast` Fortran option; `-xdepend` must be added explicitly for C programs.

- *Utilize profile feedback*

The next step is to obtain frequency information about functions in the program using the `-xprofile=collect` compile option. The application must then be executed to gather the information. Once obtained, the `-xprofile=use` option can be employed to enable the compiler to optimize the source code based on the information gathered.

- *Select floating-point optimization*

Utilizing the `-fsimple=1` (moderate) or `-fsimple=2` (aggressive) options, developers can allow the optimizer to assume less stringent rules governing numerical behavior.

- *Try `tcov` or `LoopTool`*

To obtain help in understanding the performance of an applications, developers can compile the application for `tcov` analysis using the `-xprofile=tcov` option. The resultant statement-by-statement profile of the source code will identify which statements are executed and how often. `LoopTool` can also be used to identify the amount of time spent executing each loop.

- *Try `crossfile inlining`*

Normally, the scope of the compiler's analysis is limited to each separate file on the command line. Developers can augment this functionality by enabling inlining optimizations across source files using the `-xcrossfile` option during compilation. To take advantage of this functionality, developers need to provide multiple source code files during compilation (Figure 2-21).

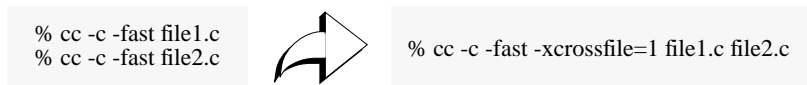


Figure 2-21 Using crossfile inlining, developers can further enhance application performance

Developers should note that `-xcrossfile` is only effective when used with the `-xO4` or `-xO5` optimization level options.

- *Utilize `global scheduling`*

The compiler can aggressively perform transformations using the `-xO5` compile option, enabling global scheduling without speculation on loads. This option should be used in conjunction with profile directed feedback to achieve the best results.

- *Use `performance pragmas`*

Performance pragmas, such as `pipelooop` and other C pragmas, enable developers to provide the optimizer with more information than a static program analysis can provide, potentially increasing application performance.

- *Enable UltraSPARC-specific instructions (UltraSPARC only)*

To take advantage of the subset of Version 9 instructions which are available to 32-bit programs, the `-xarch=v8plus` option must be used. Furthermore, to take advantage of UltraSPARC-specific instructions as well as the Visual Instruction Set, applications must be compiled with an architecture type of `-xarch=v8plusa`, `-xarch=v8plusb`, `-xarch=v9a` or `-xarch=v9b`.

- *Utilize UltraSPARC prefetch capabilities (UltraSPARC-II and UltraSPARC-III only)*

To take advantage of the prefetch capabilities of the latest UltraSPARC processors, applications must be compiled with the `-xprefetch` option. The `-xprefetch` option can only be used in conjunction with the `-xarch=v8plus`, `-xarch=v8plusa`, `-xarch=v8plusb`, `-xarch=v9`, `-xarch=v9a` or `-xarch=v9b` options, or any `-xtarget` platform that includes these architectures.

- *Utilize non-faulting loads (UltraSPARC only)*

In addition, developers may wish to try global scheduling with non-faulting loads. To do so, add `-xsafe=mem` to the list of compile options.

- *Select a good processor and architecture combination*

The final step is to ensure the application has been compiled with the target system in mind. Selecting the proper processor (`-xchip`) and instruction set type (`-xarch`) can yield significant results. Note that these must be specified after the `-fast`, `-native` (or `-xtarget=native`), and `-xtarget` options during compilation.

Multiprocessor Environments

In addition to the techniques used for uniprocessor systems, developers can take a number of steps to tune applications for multiprocessor environments:

- *Compile for fast code*

The first step toward achieving better performance is to compile the application for fast code using the `-fast` and `-xautopar`, or `-fast` and `-xparallel` options.

- *Try reduction*

Reduction operations reduce the elements of an array into a single value and can be parallelized to speed execution. By setting the `-xreduction` option during compilation, developers can direct the optimizer to generate faster code at the cost of lost determinism.

- *Ask the compiler to attempt more loop parallelizations*

The C programming language allows pointers to overlap; in Fortran this is not allowed to occur. The possibility for pointers to be overlapping often inhibits loop optimizations in C applications. By adding the `-xrestrict=function_name` option when compiling C applications, the potential exists for more loops to be parallelized by the compiler, thereby increasing application performance. Note that applications that do have overlapping pointers will experience incorrect program behavior if this option is used.

- *Obtain parallelization status information*

The `-xloopinfo` and `-xvpara` options can be used to help identify which loops could benefit from parallelization. Changes to source code can then be made to parallelize more loops.

- *Insert directives*

Source code which has not been parallelized automatically by the compiler can be directed to do so with the insertion of directives that force explicit parallelization. Once the directives have been added, the source code must be compiled with the `-xexplicitpar` or `-xparallel` options.

- *Configure the runtime environment*

Developers should remember to set the `PARALLEL` environment variable before running the application to ensure it takes advantage of multiple processors.

- *Minimize the use of pointers*

To allow the compilers greater success in discovering parallelism, developers should use arrays rather than pointers to the extent their applications permit.

Common Pitfalls

Whether designing an application for uniprocessor or multiprocessor environments, or constructing single- or multi-threaded 32- or 64-bit applications, developers can improve application accuracy and performance by properly employing tools and following a few short guidelines:

- *Select the right set of compiler flags and options*

Using the right compiler flags is critical to achieving high performance from applications. At a minimum, developers should use:

- the `-fast` option (or equivalent) to ensure optimized code is generated
- the `-xtarget` option to optimize code for a designated hardware platform
- the `-xarch` option to optimize code for a specified processor architecture
- the `-xdepend` option to enable dependency analysis
- the `-xO4` option to automatically inline source code

For maximum performance on UltraSPARC-based systems, developers can use these options as specified in Figure 2-22. The hardware options (`-xtarget`, `-xarch`) will need to be modified for maximum performance compilations for other systems.

```
% cc -fast -xtarget=ultra -xarch=v9a -xsafe=mem -xdepend
```

Figure 2-22 Using the right options can lead to increased application performance on 64-bit UltraSPARC systems running the Solaris 7 or Solaris 8 operating environment

- *Declare variables appropriately to ensure portability between 32- and 64-bit environments (Fortran)*

Two Fortran functions, `LONGJMP(env, val)` and `SETJMP(env)`, require an 8-byte integer (`INTEGER*8`) array `env[12]` when compiling for 64-bit environments. To enable portability across 32- and 64-bit environments, developers should take care to declare the `env[12]` array to be `INTEGER*8` in both cases.

- *Use compiler flags in the proper order*

The order in which compiler options are specified is an important factor in ensuring the desired compilation techniques are utilized. Flags which are specified later on the compiler command line can supersede earlier flags. For example, specifying `-xO2` prior to `-fast` will yield `-xO5`, since `-xO5` is

included in `-fast`. To ensure `-xO2` is used in combination with `-fast`, `-xO2` must be placed after `-fast` on the compile line. Consequently, developers should take care to order flags such that the desired set of compilation options is achieved. A good rule of thumb is to place `-fast` first in the compile sequence.

- *Use `-xO5` with `-xprofile=use`*

The `-xO5` optimization level option is best used in conjunction with the `-xprofile=use` option. Use of the `-xO5` option alone can result in poorer performance than when the two are used in combination.

- *Use VIS-enabled UltraSPARC instructions whenever possible*

The use of `-xarch=v9a`, `-xarch=v9b` or `-xarch=v8plusa`, `-xarch=v8plusb` may result in significantly faster applications. However, developers should be aware that applications compiled in this manner are not binary compatible with older or non-UltraSPARC-based systems.

- *Order libraries appropriately*

The order in which libraries are linked can be critical to proper application operation as well as performance. The first library in which a referenced item is found is automatically linked in to the application. This may not be the desired result, particularly if optimized versions of the library are available. For example, `-lmopt` should precede `-lm` if `-lmopt` is specified. Developers should carefully determine the order in which libraries should be linked and modify the link order, if necessary.

- *Link to proper versions of libraries*

The convention of most linkers is to ensure an application runs, rather than to make it run fast. On Sun systems, applications can be compiled and linked for a specific hardware version (SPARC Version 7, 8, 8+, 9). To ensure the proper version of the library is included, developers should take care to explicitly link in libraries with the `-L` option. In addition, the `LD_LIBRARY_PATH` runtime environment variable can be used to alter path search order.

In addition, applications compiled with the Forte Developer 6 compilers must be linked with the version 6 Sun libraries, and must not be linked with earlier library versions. Programs previously compiled with earlier versions of the compilers can be linked with the version 6 libraries.

- *Utilize the right precision functions where appropriate*

The use of single-, double-, or quad-precision functions can affect program accuracy as well as performance. Applications which utilize only 32-bit datatypes should be linked explicitly with single-precision math functions. Those which employ 64-bit or 128-bit datatypes should be linked with the double- or quad-precision functions where appropriate.

- *Utilize dynamic, shared libraries for applications to be run on SPARC Version 9 systems*

Many static libraries, such as `libm.a` and `libc.a`, are not available for SPARC Version 9 processors in the Solaris 7 or Solaris 8 operating environments. Only dynamic, shared libraries (`libm.so`, `libc.so`) are provided. Developers should note that `-Bstatic` and `-dn` may cause linking errors on SPARC Version 9 and Solaris 7 and Solaris 8 environments, and instead use the dynamic libraries in these cases.

- *Build shared, dynamic libraries properly for 64-bit environments*


When building shared, dynamic libraries with `-xarch=v9`, `-xarch=v9a` or `-xarch=v9b` in a 64-bit Solaris environment, developers must specify the `-Kpic` or `-KPIC` option, as well as take care to set `-xcode` appropriately.

- *Check system resource limits for shell environments*

For programs that create large files or use the 64-bit address space, it is important to check the system resource limits for the shell in which the application will run. In particular, the maximum file size, stack size, and data heap size should be set to *unlimited*. Developers should note, however, that programs compiled with the Forte Developer C, C++, FORTRAN 77 and Fortran 90 compilers are limited to a maximum stack size of 2 GB. The importance of setting the stack size larger than the default is displayed in Figure 2-23.

```
subroutine edit
real x(10000000)
print*,x(10)
end

call edit
end
```



```
> a.out
Killed
```

Figure 2-23 Failure to set the stack size larger than the default can result in a stacksize failure

Tuning Interval Arithmetic Applications

The Forte Developer 6 Fortran 90 compiler now supports interval arithmetic — a new computing paradigm that enables applications to perform computations with bounds on the errors from all sources, including input data errors, machine rounding, and their interactions. Combined with new advanced mathematics, interval arithmetic makes it possible to develop algorithms for solving nonlinear problems, such as the solution to nonlinear systems of equations and nonlinear programming.

To take advantage of interval arithmetic, developers must compile applications with the `-xia` and `-xinterval` options, enabling the compiler to recognize new language extensions and generate the appropriate code to implement interval arithmetic computations.

Developers can improve application accuracy and performance by properly employing tools and following a few short guidelines:

- *Utilize explicit parallelization*

Version 6 of the Fortran compilers does not support auto-parallelization for loops containing intervals. Explicit parallelization must be used to achieve performance gains associated with parallelization efforts.

- *Arrange expressions to be single-use expressions*

In general, narrow intervals contribute to fast convergence. Consequently, developers should arrange expressions to be single-use expressions (SUEs) to ensure that results are sharp (Figure 2-24). SUEs are expressions in which each variable occurs exactly once. Such simple algebraic expression rearrangement can dramatically impact the speed of interval routines.

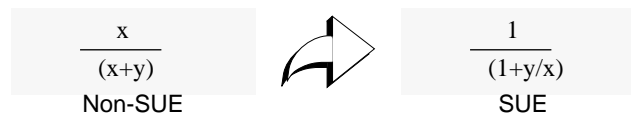


Figure 2-24 The use of single use expression ensures sharp results

Because these expressions are containment-set equivalent, containment cannot be violated when evaluated using interval arithmetic. To avoid division by an interval containing zero, the above SUE can be intersected with $1 - 1/(1+x/y)$.

More information on interval arithmetic and compiler options which support it can be found in the interval arithmetic manuals and papers listed in the references section at the end of this document.

Quality: First Class Error Detection for Single and Multi-CPU Systems

3 

Software developers know that building quality into their applications is the most cost-effective means of producing reliable code and achieving customer satisfaction. Studies have shown the earlier a software defect is found in the development cycle, the less costly it is to fix. With essential debugging tools, developers can isolate and fix problems quickly and accurately, resulting in reduced edit/integrate/build/test cycles. By addressing these problems early in the development cycle, programmers can increase application quality, decrease development costs, and react quickly to changing business needs.

Forte Developer 6 compilers include a variety of debugging tools that enable developers to provide quality solutions that meet their critical development needs, including thread-aware debugging tools, and runtime and global program checking. With these tools, application defects can be isolated, fixed, and validated in shorter timeframes. Programmers can identify and fix errors prior to application deployment, significantly decreasing the number of defects observed in the production environment.

Multithreading dbx Functionality

Sun continues to offer dbx, a time-tested, interactive debugging tool which provides facilities to run a program in a controlled fashion, and to inspect the state of a stopped program. dbx gives developers complete control of the dynamic execution of a program, including the collection of performance data. Indeed, dbx provides a variety of capabilities that enable developers to find and fix problems and increase application quality:

- View and visit code
- Control program execution

- Examine the call stack
- Evaluate and display data
- Step through a program
- Set breakpoints and traces
- Save trace output
- Manage events
- Detect runtime errors
- Modify source code
- Save and re-run a debugging run

Multithreaded features are an inherent part of the standard dbx. Developers can identify all known threads, including their current state, base functions, and current functions. In addition, thread stack traces can be examined. To ensure proper execution, application programmers can debug threads by stepping through or over a thread, navigate between them, and resume execution at any time (Figure 3-1).

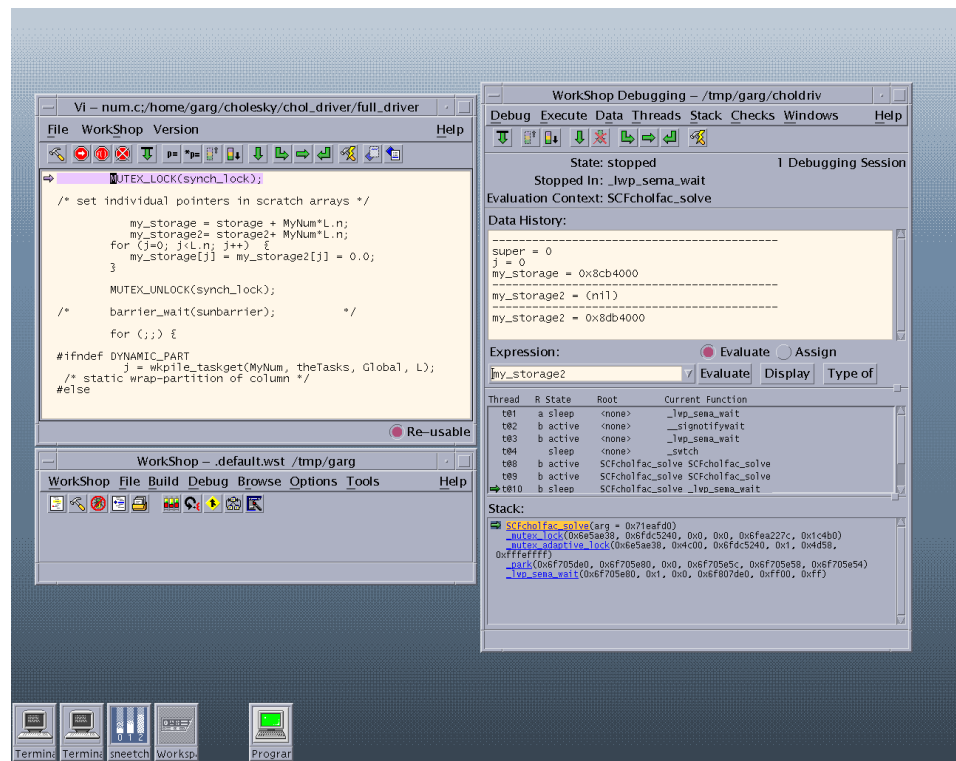


Figure 3-1 The dbx utility enables developers to find and fix problems and increase the quality of both single- and multi-threaded applications

Developers who wish to debug their multithreaded applications with `dbx` must include the `-lthread` or `-lpthread` option at link time.

More information on how to use `dbx` can be found in the *Debugging a Program with dbx* manual, part of the Forte Developer 6 Collection located at <http://docs.sun.com> on the World Wide Web.

Batch Debugging

Developers spend a significant amount of time debugging applications and need tools that facilitate that process. Indeed, the ability to locate critical debug information in object files can reduce defect isolation and resolution timeframes and improve application quality. While many developers have grown accustomed to graphical user interface-based tools, other command line driven batch utilities have proven useful. The `dumpstabs` utility is one such tool, providing developers with a batch utility for dumping out debug information.

The `dumpstabs` utility enables developers to:

- Dump the object header file
- Dump the program and section table
- Dump the debugging stabs
- Dump the linker symbols
- Dump the dynamic linking information
- Determine compile options

Each program contains information about how it was generated. Developers can choose to dump all the debug information for a program or limit the output to the debugging stabs for a specified object file. To determine the options with which a program was compiled developers can search the `dumpstabs` output for the `cmdline` keyword (Figure 3-2).

```
% cc -fast -xO4 -xparallel -xtarget=ultra example.c -o example
%
% dumpstabs example
%
% dumpstabs -s mach.o | grep -i cmdline

% f77 -o example.x -fast -xO4 -xparallel -xtarget=ultra example.f
%
% dumpstabs example.x
%
% dumpstabs -s mach.o | grep -i cmdline
```

Figure 3-2 The `dumpstabs` utility provides developers with debug information

Runtime Checking

Debugging applications is often time consuming and difficult, particularly when memory-related defects are encountered. Application developers need an efficient mechanism to detect elusive memory access violations and memory leaks that does not require the source code to be instrumented or increase its size. To aid this effort, Sun integrates on-demand runtime checking (RTC) with its debuggers, which helps to ensure the development of high quality single threaded and multithreaded applications.

RTC enables developers to automatically detect runtime errors in an application during the development phase. As errors are detected, the debugger interrupts program execution and displays the relevant source code, enabling the developer to fix bugs as they are found — greatly improving programmer productivity and application quality. Interactive filtering and scoping provides detailed control over the errors reported, resulting in more efficiently focused debugging efforts.

RTC provides a host of features that enable application developers to detect memory access errors, detect memory leaks, monitor memory usage, and debug multithreaded code.

Detect Memory Access Errors

RTC checks whether an application accesses memory correctly by monitoring each read, write, and memory free operation. RTC maintains a table that tracks the state of each block of memory being used by the program. When the program performs a memory operation, RTC checks the operation against the state of the block of memory it involves, to determine whether the operation is valid. RTC detects the following memory access errors:

- Read from uninitialized memory
- Read from unallocated memory
- Write to unallocated memory
- Write to read-only memory
- Misaligned read
- Misaligned write
- Duplicate or bad free
- Misaligned free
- Out of memory

Detect Memory Leaks

Memory leaks result in increased virtual memory consumption and generally result in memory fragmentation, resulting in poor application and system performance. Since they gradually degrade performance over time, finding them prior to application deployment is especially important.

A memory leak is a dynamically allocated block of memory that has no pointers pointing to it anywhere in the data space of the program. Such blocks are orphaned memory. Because there are no pointers to the blocks, the program cannot reference them, much less free them. Forte's RTC finds and reports such blocks.

Memory Monitor

It is often claimed that most program failures can be attributed to memory management — the dynamic allocation and deallocation of memory. Applications that fail to free objects that are no longer in use cause memory leaks. As leaked memory accumulates, application and system performance degrades. As this trend continues, the system swaps constantly and eventually runs out of memory. Other memory errors occur when objects are allocated with the wrong size, resulting in memory overwrites and erratic program behavior.

Improper memory management is a severe problem for a variety of reasons:

- Programmers typically spend 20 to 40 percent of development and maintenance time on memory management
- Memory errors are the most common form of programming errors; most programs today ship with dozens, if not hundreds of memory errors
- Memory errors are often intermittent, difficult to find and fix, and may not be detected during testing
- Memory consumption is often the most important factor in limiting computer performance

The Forte Developer 6 Memory Monitor provides a unique and complete solution for these problems. A new kind of tool that completely solves the problems of memory management in C and C++, the Memory Monitor provides a host of benefits:

- Web-enabled debugging, providing real-time reporting on memory errors and heap profiling from within a browser or Visual C++
- Unique *deployment insurance*, automatically protecting applications against memory leaks and premature frees, even if caused by third-party libraries
- Ability to develop applications without calling *free* or *delete*, bringing one of the major benefits of Java to C and C++ programmers
- Unique heap profiler, proving monitoring of memory usage from within a browser or Visual C++
- Detailed statistics that identify exactly how much memory each part of a program is using at any time
- Exclusive non-fragmenting, high-performance allocator, enabling programs to run faster in less space and never fragment memory.

To take advantage of Memory Monitor capabilities, developers must link applications with the Forte Developer Memory Monitor libraries to automatically manage all of a program's memory. These libraries are available in both deployment (`libgc.a` or `libgc.so`) and development (`libgc_dbg.a` or `libgc_dbg.so`) mode versions. Simply linking with the `libgc` library automatically and permanently fixes program memory leaks, and enables developers to write applications without calling `free()` or `delete()` while otherwise programming normally.

More information on the Memory Monitor can be found in the *Memory Monitor Libraries* section of the on-line manuals.

Global Program Checking

To facilitate the debugging of Fortran applications, Sun's Fortran compilers include global program checking (GPC). Invoked with a `-Xlistx` option, GPC provides a valuable way to analyze a source program for inconsistencies and possible runtime problems:

- Stringently enforces Fortran type-checking rules
- Enforces portability restrictions needed to move programs between systems
- Detects legal constructions that may be suboptimal or error-prone

In particular, global program checking reports problems in a number of areas, including interface, usage, and syntax. GPC detects conflicts in the number and type of arguments, wrong types of function values, and possible conflicts due to data type mismatches in common blocks between different subprograms. Usage

problems reported include functions used as subroutines, subroutines used as functions, and declared but unused functions, subroutines, variables and labels. In addition, GPC isolates unreachable statements, syntax errors, and portability issues that arise from code that does not conform to ANSI Fortran.

Developers interested in using global program checking to debug Fortran applications and ensure application quality should become familiar with all `-Xlist` suboptions, as listed in Table 3-1.

Option	Action
<code>-Xlist</code> (no suboption)	Show errors, listing, and cross-reference table
<code>-Xlistc</code>	Show call graphs and errors
<code>-XlistE</code>	Show errors
<code>-Xlisterr[nnn]</code>	Suppress error nnn in the verification report
<code>-Xlistf</code>	Produce fast output
<code>-Xlistfindir</code>	Put the .fn files in dir
<code>-Xlisth</code>	Errors from cross-checking stop compilation
<code>-Xlistl</code>	List and cross-check include files
<code>-XlistL</code>	Show the listing and errors
<code>-Xlistln</code>	Set page breaks
<code>-Xlisto name</code>	Rename the <code>-Xlist</code> output report file
<code>-Xlists</code>	Unreferenced symbols suppressed from the cross-reference
<code>-Xlistvn</code>	Show different amounts of semantic information
<code>-Xlistw[nnn]</code>	Set the width of output lines
<code>-Xlistwar[nnn]</code>	Suppress warning nnn in the report
<code>-XlistX</code>	Show the cross-reference table and errors

Table 3-1 Sun Fortran compilers include a variety of global program checking options

More information on global program checking can be found in the *Forte Developer 6 Compiler Fortran Collection* at <http://docs.sun.com> on the World Wide Web.

Analyzing C Source Code — LockLint

Deadlock and race conditions are often difficult to locate and correct, and developers need tools that can aid this effort. LockLint is one such tool, giving developers the ability to statically analyze C source code and detect data races and deadlock conditions caused by inconsistent usage of mutex and reader-writer locks. With LockLint, developers can create scripts and annotate source code to help them quickly and effectively locate programmatic errors and resolve them.

Indeed, LockLint enables developers to perform a wide range of functions, including:

- Analyze the loaded files for lock inconsistencies which may lead to data races and deadlocks via the analyze command. Developers may wish to redirect output to a file as this action produces a large quantity of results.
- Determine which variables are not properly protected by locks
- Provide LockLint with assertions about how locks are being used, enabling LockLint to report on any violations of these assertions during analysis
- List calling sequences which are not permitted
- Declare lock, function, and variable attributes
- Obtain information about the functions and function pointers used in the loaded files
- Exclude functions and variables from analysis
- Obtain information about the locks on loaded files
- List the members of structures with specified tags
- Determine the order in which locks are acquired by the code under analysis
- List calls made through function pointers
- Allow exceptions to disallow commands
- Refresh and restore the stack
- Save the current state of LockLint on a last-in-first-out (LIFO) stack

To take advantage of LockLint, developers must compile C applications with the `-Zll` option. By employing `-Zll`, developers instruct the compiler to produce a LockLint database file for each C source file compiled. The resulting database files (named with a `.ll` extension) can be loaded into LockLint for analysis.

More information on LockLint can be found in the *Analyzing Program Performance with Forte Developer* manual list in the references section at the end of this document.

Productivity: Intuitive, Efficient, and Extensible Application Development Using Forte Developer 6 Products

4 

For developers to be efficient, they must be able to seamlessly perform all programming tasks, including editing, debugging, testing, and profiling within an integrated environment. Using Sun systems and integrated development environments, programmers can now work concurrently on a variety of tasks.

The multitasking features of Sun's Solaris operating system and graphical windowing system enable developers to simultaneously modify files, parallelize efforts, and decrease compile and link times. By enhancing the Solaris environment with integrated, powerful, visual developer tools, programmer productivity is further enhanced — corporate development bottlenecks can be alleviated through effective, efficient, and flexible team application development.

The Forte Developer 6 suite offers a host of features that facilitate developer productivity:

- Integrated programming environment
- Motif user interface, providing a standard look and feel
- Tight, editor-centric tool integration
- Hyperlinks, enabling easy tool navigation
- Multiprocessing, multithreaded development tools
- Distributed and parallel make utilities
- Incremental linker, for faster builds
- Fix and Continue, enabling defects to be found and fixed quickly
- AppGuru, enabling very fast application development for C++
- New version of Rogue Wave Tools.h++ 7.0 class library
- Motif, Windows, and Java GUI Builder, for cross-platform development
- Project Manager, for coordinating files, tools and targets for a project

- Forte C++ Enterprise and Personal Editions, enabling quick and easy GUI development
- GUI capture and testing, providing reverse engineering capabilities
- Three dimensional data visualizer, speeding debugging of complex arrays
- WorkSets and PickLists, facilitating quick access to work sessions
- Forte TeamWare, for source code and configuration management
- Extensive on-line manuals and help system

This chapter focuses on the new, innovative, powerful, or upgraded Forte Developer products and features that enhance developer productivity.

Integrated Programming Environment

The latest release of the Forte Developer application development product family offers a new, integrated programming environment. Its editor-centric, Motif-standard user interface integrates all the steps programmers take — from GUI design and code generation to edit-compile-debug-tune cycles — making it easy to rapidly build high performance applications (Figure 4-1).

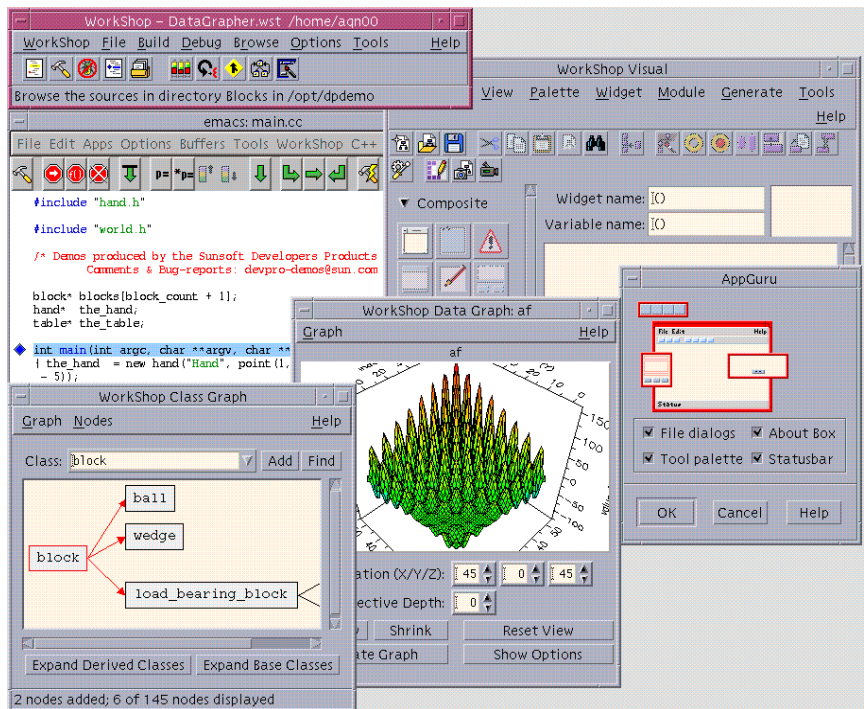


Figure 4-1 Forte Developer’s integrated programming environment enhances developer productivity

Not just a compiler suite, Forte Developer 6 integrates several key tools that enable the efficient development of high quality applications:

- Choice of integrated editors (vi, XEmacs, GNUemacs, Gvim, Nedit)
- Project Manager
- Data visualization techniques
- Fix and Continue debugging features
- Incremental linker to shorten build times
- GUI builder for Visual C environments
- Extensive on-line HTML-based help and documentation

Integrated Editors

No two developers write and edit code the same way. The Forte Developer 6 text editors form the center of the integrated development tool set that includes building, debugging and browsing. The Forte Developer integrated editors make it possible for programmers to evaluate expressions, set breakpoints, and step through functions from a familiar text editor.

Forte Developer 6 includes the following integrated editors:

- *NEdit* version 5.0.2, a graphical user interface-style plain-text editor for X/Motif systems. NEdit is the default Forte Developer editor. More information about NEdit can be found at <http://www.nedit.org> on the World Wide Web.
- *XEmacs* version 20.4 (or compatible versions), a customizable text editor and application development system. More information on XEmacs can be found at <http://www.xemacs.org> on the World Wide Web.
- *GNU Emacs* version 19.28 (or compatible versions), an extensible, customizable, self-documenting real-time display editor. More information on GNU Emacs can be found at <http://www.gnu.org> on the World Wide Web.
- *Vi*, a popular screen-based editor on UNIX systems.
- *Vim* version 5.3 with graphical user interface option, an improved version of the `vi` standard text editor on UNIX systems. More information on Vim can be found at <http://www.vim.org> on the World Wide Web.

Tool Integration

The Forte Developer 6 suite integrates all editors with the programming environment. With this release, several new tools facilitate the debugging and editing of source code:

- *Warp-to-source*, enabling developers to see and modify errors regardless of the editor being used. As developers click on errors in the build output window, hyperlinks automatically position the cursor to the source in the editor. As changes are made, error line number adjustments occur automatically, enabling developers to quickly and easily find the next problem to be fixed.
- *Balloon expression evaluator*, enabling developers to see instantly the current value and type of the expression pointed at in the text editor, as well as dereference pointers.

Project Manager

The Project Manager enables developers to keep track of files, programs, and targets associated with development projects. With Project Manager, developers can build applications without the need for a traditional makefile. Projects are lists of files and compiler, debugger, and build related options used to construct an executable, a static library or archive, a shared library, a Fortran application, a complex application, or a user makefile application. Project Manager enables developers to:

- Create and edit projects
- Build project targets
- Select text editors for projects
- Configure startup and project options
- Access tools, including compilers, source code browsers, debuggers, analyzers

More information on the Project Manager can be found in the *Introduction to WorkShop* manual located at <http://www.docs.sun.com>.

Data Visualization

Scientific and numerical software developers work with large volumes of data. To facilitate their analyses, they need to “see” data. *Data visualization* is a debugging technique that enables developers to explore and comprehend large and complex datasets, simulate results, and interactively steer computations. It has become a vital research and applications frontier with impact on the

scientific, engineering, medical, business, and entertainment markets. These demanding environments mandate the ability to navigate data, animate simulations, interact with data, and “see” the results.

The Forte Developer debugger enables developers to view their programs and visualize data. During this process, developers can update the data on demand, at specified breakpoints, or at specified time intervals. The tools provided enable changes to be observed as well as analyzed (Figure 4-2).

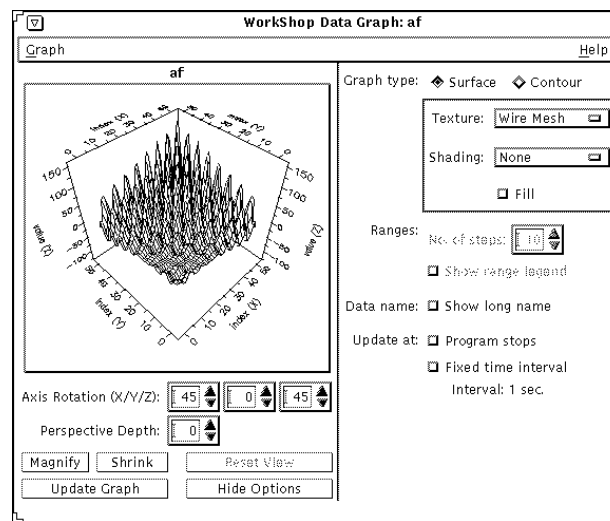


Figure 4-2 Sun's data visualization tools give developers the ability to “see” program data and analyze that data graphically

Fix and Continue

Once an error has been identified, developers need a mechanism that enables it to be corrected quickly without disrupting the debugging session. The Fix and Continue feature included in Forte Developer debuggers automatically compiles source code changes and modifies the executing process with the fix. Developers can then resume the debugging process to confirm the fix worked as desired. This rapid turnaround eliminates the often lengthy link phase and the amount of time taken to restart the debugger and recreate the debugging session. With Fix and Continue, incremental software development in a production development environment becomes a reality.

Fix and Continue Operation

When a source code file is modified, it is compiled and a shared object file is created. Semantic tests are done by comparing the old and new files. The new object file is linked to the running process using the runtime linker. If the function on top of the stack is being fixed, the new *stopped in* function is the beginning of the same line in the new function. All the breakpoints in the old file are moved to the new file. The modified source replaces the old source in the editor window, and the developer can resume debugging from the exact point where he stopped.

Incremental Linker

Sun's incremental link editor (`ild`) allows developers to complete the edit-compile-link-debug cycle more quickly than by using a standard linker. It accelerates the rebuilding of a large application after a source code change has been made.

To speed the link process, `ild` incorporates additional space into the initial build. During subsequent builds, newly changed information is stored in the additional space. If insufficient space exists, a complete new link is generated. While these operations are transparent to the developer, subsequent builds may be observed to be significantly faster than the original.

Developers should note that `ild` generates significantly larger binaries than conventional linkers. When compiling with the `-g` option, `ild` is on by default, and off otherwise.

Forte C++ Enterprise and Personal Editions

Research shows that user interface code can comprise up to seventy percent or more of an application's source code base. A significant portion of this code is considered standard — all applications include it or something similar. The graphical user interface builder in the Forte C++ Enterprise and Personal Edition environments generates this common code automatically, significantly reducing application development time and increasing code reliability and quality.

A time saving GUI development tool, Forte C++ enables sophisticated GUI applications to be built quickly and easily from the ground up or by reverse engineering existing applications. Its highly graphical environment supports application construction through a point-and-click mechanism for building and

connecting widgets. When the layout is complete, Forte C++ generates portable C or C++ GUI code automatically and ensures all interfaces are fully compatible with OSF/Motif. Figure 4-3 illustrates the intuitive user interface capabilities of Forte C++.

Highlights of the Forte C++ Enterprise and Personal Editions include:

- Powerful, easy-to-use GUI designer
- Compliant with industry standards, including X11 R5 and OSF/Motif 1.2
- Non-modal interface enables concurrent build and test cycles
- Extensible palette, including standard, custom and third-party widgets
- Full access to the Motif API
- Layout Editor supports easy design layout
- Compound String Editor
- Font Editor
- Color Pixmap Editor supports XPM, XPM3 and X bitmaps
- Search and annotate functions
- Cross-platform development

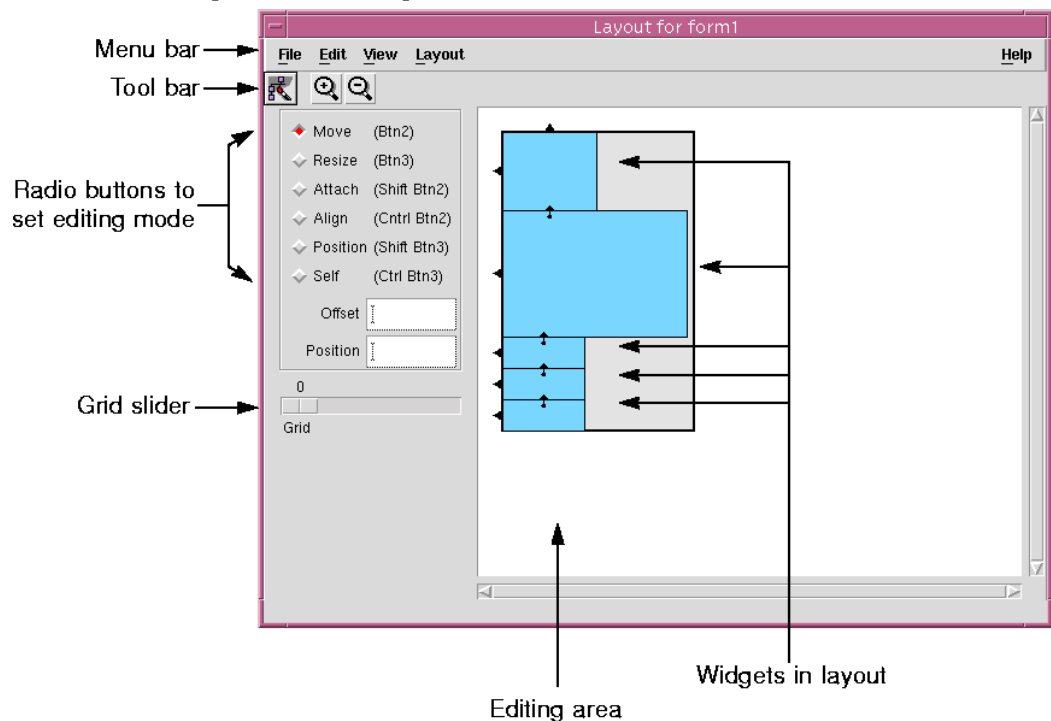


Figure 4-3 Forte C++ Enterprise and Personal Editions provide an intuitive tool that eases GUI construction

Integrated make Utility — Building Window

The Forte Developer 6 suite includes a facility for building applications, eliminating the need for developers to step outside the integrated environment to perform a task. From the Building Window, developers can stop a build in progress, edit build parameters, save the build output, and view build errors — commonly performed tasks that have been integrated into a coherent whole (Figure 4-4).

More information on Forte Developer can be found in the *Using Forte Developer* manual located at <http://www.sun.com/workshop> on the World Wide Web.

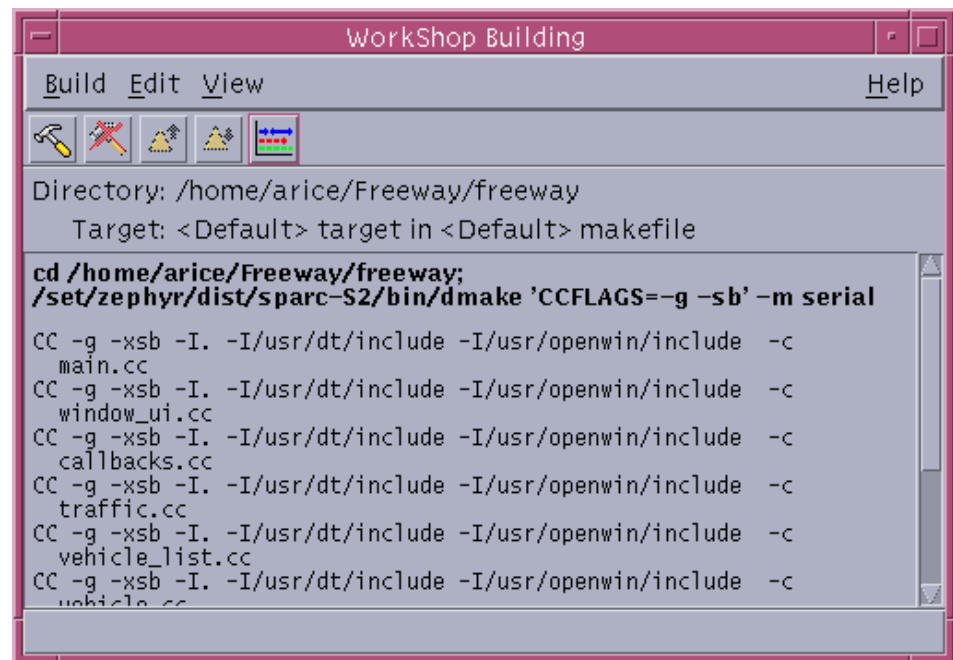


Figure 4-4 The Building Window integrates common make functions

Remote Monitoring

The nature of distributing computing and application deployment raises additional concerns for developers. While integrated development environments provide an intuitive platform for constructing and debugging local applications, programmers can also configure the computing platform to enable them to assist

in the remote monitoring and debugging of applications. This is of particular importance in environments where the development, test, and deployment systems are geographically distant or have significantly dissimilar hardware or software configurations.

In order to take advantage of the benefits of remote monitoring, developers must login remotely to the system running the application and set its display to the local machine. These steps enable the application to run on the remote system and display on the local system, providing developers with the ability to watch the application in action without forfeiting the use of their sophisticated debugging tools (Figure 4-5).

```
local-host# xhost +remote-host
local-host#
local-host# rlogin remote-host
remote-host# ...
remote-host# setenv DISPLAY local-host:0.0
remote-host#
remote-host# program-name
```

Figure 4-5 Remote monitoring enables developers to watch applications run on remote systems yet use local development and debugging tools

Accessing Sun Documents On-line

The docs.sun.com Web site enables developers to access Sun technical documentation online. Developers can browse the docs.sun.com archive or search for a specific book title or subject. The following Forte Developer documentation is available online at <http://www.docs.sun.com>.

- *Forte Developer 6 manuals*, available in HTML format or as printed material
- *Online help*, available in HTML format
- *Man pages*, available in HTML or ASCII format
- *Component readme files*, available in HTML or ASCII format

Summary

5 

Computers are an integral part of today's business environment, and applications are constantly challenged to meet performance, quality, and feature requirements. To meet tight development schedules and simultaneously produce high quality, high performance applications, programmers need sophisticated tools that aid the development of optimized and parallelized 32- and 64-bit applications. They also need powerful debugging and analysis tools. When used in combination, such tools enable the development of quality solutions that meet the needs of today's high performance environments.

The Forte Developer 6 product family is a powerful, visual application development environment that incorporates all the tools developers need to create high quality, high performance, single and multithreaded 32- and 64-bit applications. Offering optimized compilers for C, C++, and Fortran, browser, debugger, performance analyzer, and software configuration management tools, the Forte Developer family gives developers an integrated environment for rapidly building high performance applications.

Sun Microsystems understands that far more than processor superiority is needed to ensure success. Sun has carefully crafted a highly integrated distributed computing architecture that includes powerful, binary compatible desktop and server systems, an operating environment that incorporates key industry standards, and software products that enable the construction of powerful applications for these systems.

Glossary



Cache blocking

A technique to rearrange loop code to make maximum use of the processor cache.

Concurrent processes

Processes that execute in parallel on multiple processors or asynchronously on a single processor. Concurrent processes may interact with each other, and one process may suspend execution pending receipt of information from another process.

Dead code elimination

A compiler technique that eliminates unreachable code.

dumpstabs

A utility for dumping out debug information.

Garbage collection

The act of reclaiming and consolidating unused blocks of memory.

GUI

Graphical User Interface.

HTML

Hyper Text Markup Language. A text markup language used to format documents for display and viewing by Web browsers.

Inlining

A compiler technique to replace subroutine calls with source code. This reduces overhead and takes advantage of special circumstances, such as constant parameters.

Instruction scheduling

A compiler technique that maximizes processor instruction issue strategies.

Library

A set of subprograms that have been previously compiled and organized into a single binary library file. Each member of the set is called a library element or module. The linker searches the library files, loading object modules referenced by the user program while building an executable binary program or at run-time.

Loop invariant hoisting

A compiler technique to evaluate expressions within a loop whose values do not change across loop iterations before loop begins execution.

Loop inversion

A compiler technique that converts pre-test loops into post-test loops, reducing the number of branches required per iteration.

Loop parallelization

A technique used by the compiler to rearrange loop code so that multiple processors may work in parallel to complete the loop.

Loop pipelining/unrolling

A compiler technique that arranges loop code to ensure idle time and idle resources during one loop iteration may be applied to another. This results in fewer loop control instructions.

Multiprocessor system

A system in which more than one processor can be active at any given time.

Multitasking

In a uniprocessor system, a technique in which a large number of tasks appear to be running in parallel. This is accomplished by rapidly switching between tasks.

Multithreading

Applications that can have more than one thread or processor active at one time are multithreaded. Multithreaded applications can run in both uniprocessor and multiprocessor systems.

Parallel processing

In a multiprocessor system, true parallel execution is achieved when a number of threads or processes can be active at one time.

Parallelism

See multithreading.

Parallelization directive (pragma)

Parallelization directives, or pragmas, comment lines that tell the compiler to parallelize (or not to parallelize) the loop that follows the directive. Pragmas enable the developer to effectively parallelize loops that might be difficult for the compiler.

SPARC Version 9

The 64-bit SPARC specification. Now available through SPARC International's SPARCShop. ISBN number 0-13-099227-5.

Strength reduction

A compiler technique that replaces slower operations with faster ones. Typical cases include replacing a multiply operation inside a loop with an addition operation, and replacing a constant multiplier with shift and add operations.

Tail recursion elimination

A compiler technique that converts self-recursive procedures into iterative procedures, saving stack manipulation time and minimizing register window spills.

Thread

A flow of control within a single UNIX process address space.

Uniprocessor system

A system with one processor. This single processor can run multithreaded applications as well as the conventional single instruction data model.

References



Sun Microsystems Computer Company posts product information in the form of data sheets, specifications, and white papers on its Internet World Wide Web Home page at: <http://www.sun.com>.

Look for abstracts on these and other Sun technology white papers and manuals:

Delivering Performance on Sun: Optimizing Applications for Clustered Systems, Sun Microsystems, 1998.

Analyzing Program Performance with Forte Developer, Sun Microsystems, 1999.

C++ Library Reference, Sun Microsystems, 2000.

C++ Migration Guide, Sun Microsystems, 2000.

C++ Programming Guide, Sun Microsystems, 2000.

C++ Standard Library 2.0 User's Guide, Sun Microsystems, 2000.

C++ Standard Library Class Reference, Sun Microsystems, 2000.

C++ User's Guide, Sun Microsystems, 2000.

Debugging a Program with dbx, Sun Microsystems, 2000.

Fortran 77 Language Reference, Sun Microsystems, 2000.

Fortran Library Reference, Sun Microsystems, 2000.

Fortran Programming Guide, Sun Microsystems, 2000.



-
- Fortran User's Guide*, Sun Microsystems, 2000.
- Incremental Link Editor*, Sun Microsystems, 2000.
- Interval Arithmetic Programming Reference*, Sun Microsystems, 2000.
- Linker and Libraries Guide*, Sun Microsystems, 1998.
- mediaLib Quick Reference Guide*, Sun Microsystems.
- mediaLib Reference Manual*, Sun Microsystems.
- Numerical Computation Guide*, Sun Microsystems, 2000.
- SPARC Strategy and Technology*, Sun Microsystems, 1991.
- Sun MP C Compiler, technical white paper, Sun Microsystems, 1997. Located at <http://www.sun.com/servers/hpc/whitepapers/> on the World Wide Web.
- Sun Performance Library User's Guide*, Sun Microsystems, 2000.
- Sun Performance and Tuning: SPARC and Solaris*, Adrian Cockcroft, Prentice Hall, October 1994, ISBN 0-13-149642-5.
- Forte Developer Memory Monitor User's Guide*, Sun Microsystems, 2000.
- Tools.h++ Class Library Reference*, Sun Microsystems, 1999.
- Tools.h++ User's Guide*, Sun Microsystems, 1999.
- UltraSPARC Programmer Reference Manual*, Sun Microsystems, 1995.
- UltraSPARC User's Manual*, Revision 2.0, Sun Microsystems, June 1996, part number 802-7220-01.
- The UltraSPARC Processor*, Sun Microsystems, July 1997.
- The UltraSPARC-III Processor*, Sun Microsystems, 1999.
- Using Sun Performance Library*, Sun Microsystems.
- Using Forte Developer*, Sun Microsystems, 1999.
- VIS Instruction Set User's Guide*, version 1.1, Sun Microsystems, March 1997, part number 805-1394-01.



Documents from other sources:

LAPACK User's Guide, 2nd edition, SIAM, 1996.

LINPACK User's Guide, SIAM, 1996.

Web sites of interest:

<http://access1.sun.com/workshop>

<http://docs.sun.com>

<http://java.sun.com/docs>

<http://www.sun.com/software/whitepapers.html>

<http://www.sun.com/solaris/opengl>

<http://www.sun.com/sparc/vis>

<http://www.sun.com/sparc/whitepapers>

<http://www.sun.com/sun-on-net/mikes/html>

<http://www.sun.com/workshop/sitemap.html>

<http://www.sun.com/workshop/workshopFAQ.html>

<http://www.sunsite.unc.edu>

<http://www.validgh.com/reports>

<http://www.SGI.com/Technology/openGL>





Sun Microsystems, Inc. <http://www.sun.com>

Sales Offices

Argentina: +54-1-311-0700
Australia: +61-2-9844-5000
Austria: +43-1-60563-0
Belgium: +32-2-716-7911
Brazil: +55-11-524-8988
Canada: +905-477-6745
Chile: +56-2-638-6364
Colombia: +571-622-1717
Commonwealth of Independent States: +7-502-935-8411
Czech/Slovak Republics: +42-2-205-102-33
Denmark: +45-44-89-49-89
Estonia: +372-6-308-900
Finland: +358-0-525-561
France: +33-01-30-67-50-00
Germany: +49-89-46008-0
Greece: +30-1-680-6676
Hong Kong: +852-2802-4188
Hungary: +36-1-202-4415
Iceland: +354-563-3010
India: +91-80-559-9595
Ireland: +353-1-8055-666
Israel: +972-9-956-9250
Italy: +39-39-60551
Japan: +81-3-5717-5000
Korea: +822-3469-0114
Latin America/Caribbean: +1-415-688-9464
Latvia: +371-755-11-33
Lithuania: +370-729-8468
Luxembourg: +352-491-1331
Malaysia: +603-264-9988
Mexico: +52-5-258-6100
Netherlands: +31-33-450-1234
New Zealand: +64-4-499-2344
Norway: +47-2218-5800
People's Republic of China:
Beijing: +86-10-6849-2828
Chengdu: +86-28-678-0121
Guangzhou: +86-20-8777-9913
Shanghai: +86-21-6247-4068
Poland: +48-22-658-4535
Portugal: +351-1-412-7710
Russia: +7-502-935-8411
Singapore: +65-224-3388
South Africa: +2711-805-4305
Spain: +34-1-596-9900
Sweden: +46-8-623-90-00
Switzerland: +41-1-825-7111
Taiwan: +886-2-514-0567
Thailand: +662-636-1555
Turkey: +90-212-236-3300
United Arab Emirates: +971-4-366-333
United Kingdom: +44 (0)1252 420000
United States: +1-800-821-4643
Venezuela: +58-2-286-1044
Worldwide Headquarters: +1-415-960-1300