Technical White Paper

SPARCompiler Compilation Technology

Sun Microsystems, Inc.

July 1991

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Revision 1.0

© 1991 by Sun Microsystems, Inc.—Printed in USA.
2550 Garcia Avenue, Mountain View, California 94043-1100

All rights reserved. No part of this work covered by copyright may be reproduced in any form or by any means—graphic, electronic or mechanical, including photocopying, recording, taping, or storage in an information retrieval system— without prior written permission of the copyright owner.

The OPEN LOOK and the Sun Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

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 (October 1988) and FAR 52.227-19 (June 1987).

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

TRADEMARKS

The Sun logo, Sun Microsystems, Sun Workstation, NeWS, and SunLink are registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Sun, Sun-2, Sun-3, Sun-4, Sun386*i*, SunCD, SunInstall, SunOS, SunView, NFS, and OpenWindows are trademarks of Sun Microsystems, Inc.

UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc.

X Window System is a product of the Massachusetts Institute of Technology.

SPARC is a registered trademark of SPARC International, Inc. Products bearing the SPARC trademark are based on an architecture developed by Sun Microsystems, Inc. SPARCstation, SPARCompiler and SPARCworks are trademarks of SPARC International, Inc., licensed exclusively to Sun Microsystems, Inc.

All other products or services mentioned in this document are identified by the trademarks, service marks, or product names as designated by the companies who market those products. Inquiries concerning such trademarks should be made directly to those companies.

Contents

Introduction 1

SPARCompiler Family Overview 2

C, C++, FORTRAN, and Pascal 4

- Compiler Structure 5
- Levels of Optimization 11

Ada 15

- Optimizations 17
- Fast Exceptions 18
- Passive Tasks 18
- Remote Compilation 19
- Large Program Support 19
- Interlanguage Calling 19
- Miscellaneous Optimizations and Pragmas 19

Sun Common Lisp 21

- The Common Lisp Object System (CLOS) 21

Dual Compiler System 22
Optimization Reporting Facility 23
Foreign Function Interface 23
Ephemeral Garbage Collection 23
Multitasking (Stack Groups) 24
SPARC Support for Lisp 24
Sun COBOL 29
Sun COBOL Compiler and Interpreter 29
Choosing Between Interpreted, Generated, and Compiled Code 30
Dynamic Loading in Sun COBOL 31
Flags, Directives, and Switches for Sun COBOL 31
References 33
Appendix A — Optimization Definitions 34

SPARCompiler™ Compilation Technology

Introduction

This report is one in a series of white papers that describe the technical aspects of Sun's software development products. Currently there are three papers in this series:

- The *SPARCworks™ Development Environment* white paper discusses the rich software development environments offered for each product.
- The *SPARCompiler Benchmark* white paper provides detailed benchmarking information about each of the seven compiler products. [forthcoming]
- This document, the *SPARCompiler Compilation Technology* white paper, introduces the compilation technology used in each of the seven SPARCompiler products. It also highlights the products' relationship with the Scalable Processor ARChitecture (SPARC™). The rest of this white paper assumes that the reader is familiar with the basic vocabulary of compilers and computer architecture.
- In addition, many of the products such as Sun Ada and Sun Common Lisp have their own white papers.

The SPARCompiler family of robust, optimizing compilers and environments provides the cornerstone of Sun's software engineering portfolio. With seven powerful compilers — Ada, C, C++, COBOL, Common Lisp, FORTRAN, and Pascal — and the full range of SPARCsystem platforms, Sun offers you the advantage of a single source for the system and tools you need to make the most of your software development investment.

Designed in concert with SPARC technology, SPARCompiler products take full advantage of the SPARC architecture to provide optimization that delivers unprecedented performance. The code generation modules of the SPARCompiler products utilize state-of-the-art innovations in compiler technology, particularly in code optimization. Because compilers for SPARC and other Reduced Instruction Set Computer (RISC) architectures synthesize instruction sequences that correspond to Complex Instruction Set Computers'

(CISC) more complicated instructions, RISC compilers often produce more instructions (up to 20% more) than comparable CISC machines. However, these are almost all single-cycle instructions. Therefore, good optimization technology plays a very important role in SPARC system performance.

An individual SPARC machine is an implementation of the SPARC Instruction Set Architecture (ISA). The performance of a SPARCsystem is a function of the architecture, the hardware implementation, and the code generated by the compiler. Because the SPARC ISA and SPARCompiler technology were developed in concert, the compilers take careful advantage of the architecture to improve performance. Among the architectural features SPARC includes are:

- Register windows
- Delayed branches and delayed loads
- Hardware interlocks
- Floating-point coprocessors

Optimized for the SPARC architecture and hardware implementations, SPARCompiler products can significantly increase application speed and therefore play an integral role in the performance of SPARCsystems.

SPARCompiler Family Overview

The seven members of the SPARCompiler family — Ada, C, C++, COBOL, Common Lisp, FORTRAN, and Pascal — all share key features that enhance each product, enable coordination between the products, and maximize your development dollar. The major features, and their benefits, are:

Table 1 Features/Benefits

Features	Benefits
Optimized for the SPARC architecture	• Deliver unprecedented performance on SPARC platforms
Interlanguage calling	• Because they enable you to combine your existing code with modules written in other languages, SPARCompilers protect your current software investment
Multiple levels of optimization	• Provide flexibility to control compile-time versus execution-time and memory size versus space trade-offs in the compiled code
International character set support	• Meet the needs of global markets
Industry and *de facto* standards	• Ensure compatibility and portability while providing competitive advantages through language enhancements
Access to graphics and OpenWindows™ XView™ libraries, UNIX system calls, and SunOS™ enhanced utilities	• Reduce time and resources required to develop sophisticated applications and shorten time to market
Integration with SPARCworks programming tools and the OpenWindows window system	• Create an integrated development environment that enhances programmer productivity

The Sun SPARCompiler products combine lexical, syntactic, and static semantic components — the language "front ends," with code generation and optimization modules — the language "back ends." Four of the SPARCompiler products — C, C++, FORTRAN, and Pascal — share the same back end, while the other three back ends provide language specific optimizations and features. As Figure 1 on page 4 shows, all seven languages are targeted to the SPARC architecture and are supported by powerful programming environments.

The next section describes the C, C++, FORTRAN, and Pascal compilers, while the following three sections discuss the Ada, Common Lisp, and COBOL compilers.

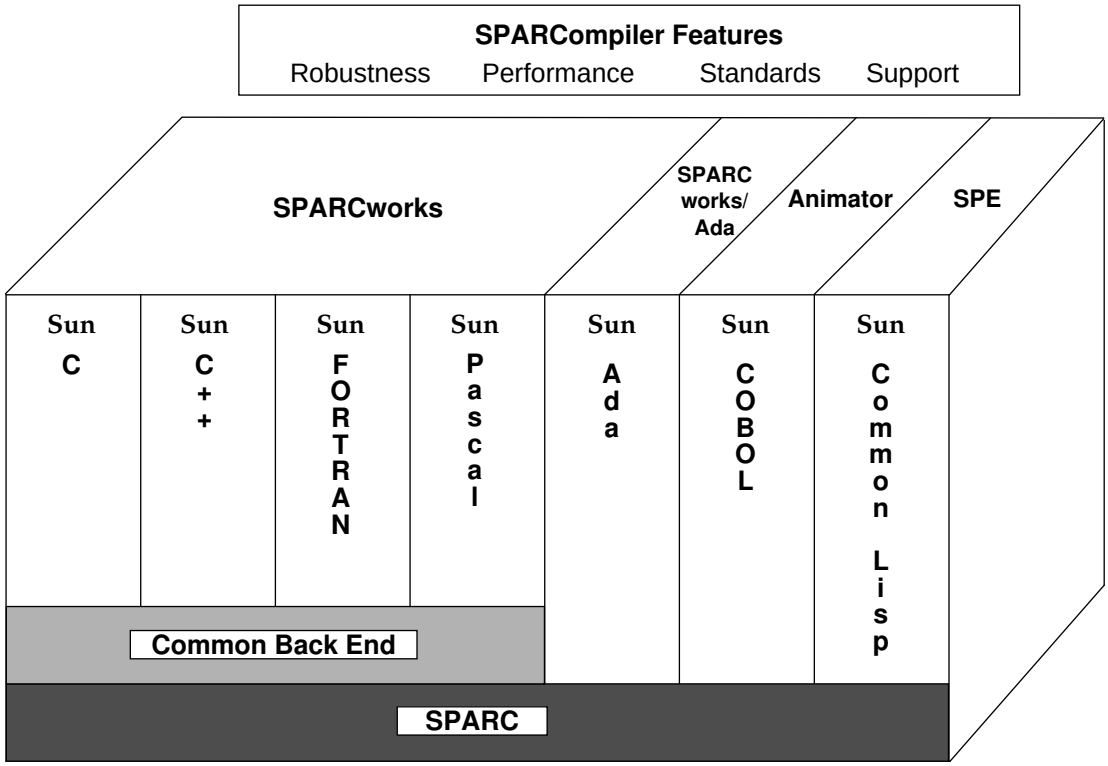


Figure 1 SPARCompiler Features

C, C++, FORTRAN, and Pascal

As mentioned above, four of the SPARCompilers — C, C++, FORTRAN, and Pascal — share a single, very efficient back end. The reasons that these particular languages share a common back end are the characteristics of the languages themselves, historical engineering effort at Sun, and the needs of the development tool environments. The major benefit of focusing engineering

efforts on a common back end is that any performance improvement or bug fix enhances all four of these compilers. The following section describes the structure of the C, C++, FORTRAN, and Pascal compilers.

Compiler Structure

Figure 2 on page 6 shows how a program flows through the compilation phases that transform it into an executable program. The solid arrows describe the path when optimization and inline code expansion are both enabled. When either is disabled, certain components are skipped.

Sun's optimization technology is designed to satisfy several goals, including:

- Support for multiple source languages
- Production of high-quality, high-performance code
- Reduction of compilation time to no more than necessary to do ambitious optimization

The rest of this section describes the various phases of the compilation process, with emphasis on optimization.

Preprocessors are programs that manipulate source text; they transform the code into a form acceptable by a compiler or assembler. `cpp` is the most widely used preprocessor. It is independent of any language (although it was designed to be used with C) and can be used to define symbolic constants, insert files into the source stream, expand macros, and conditionally compile segments of code.

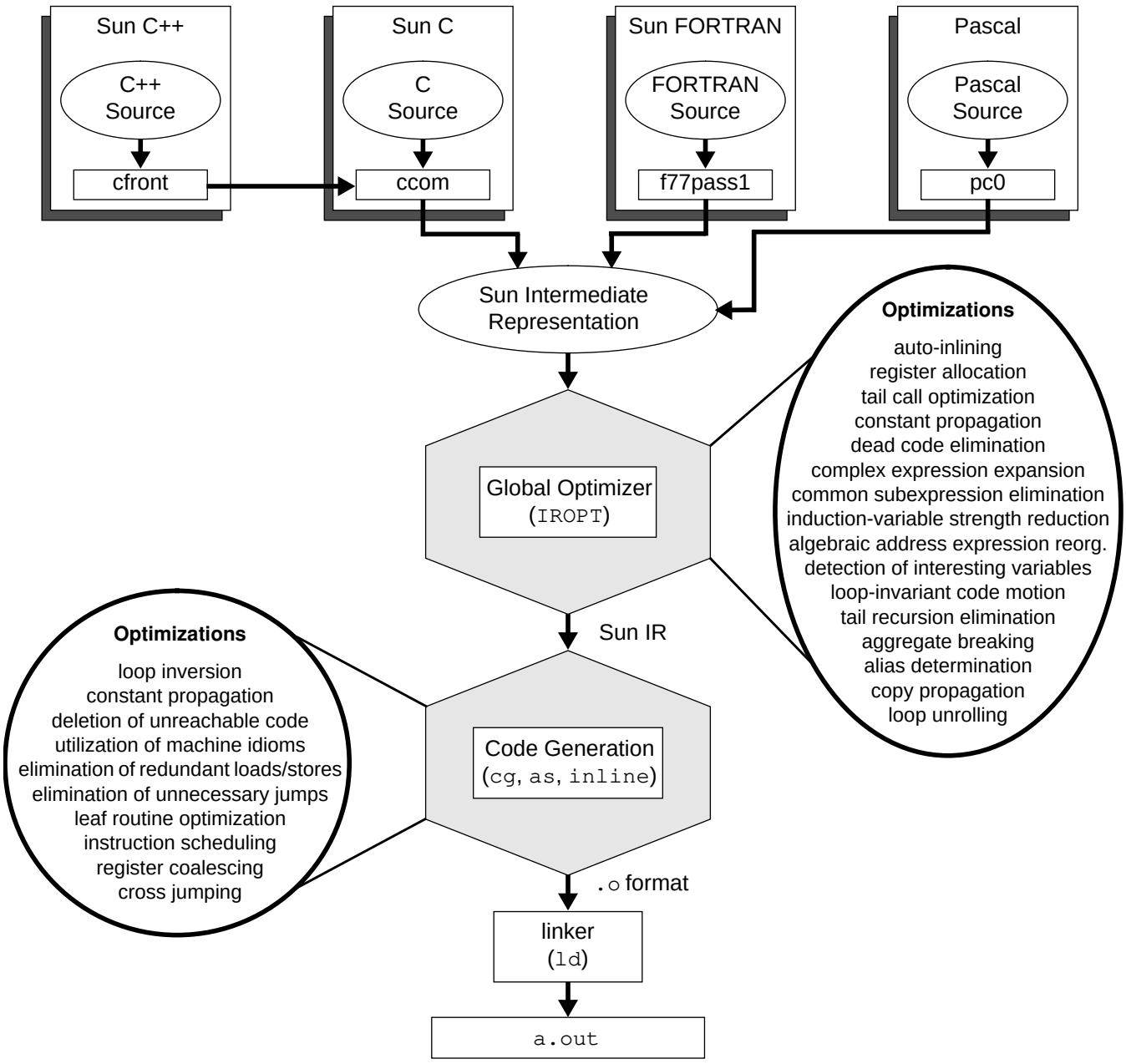


Figure 2 SPARCompiler Structure and Features

The front end scans and parses the source-language statements that constitute a procedure and checks static semantics. The target of the front end is an intermediate language called Sun IR (Intermediate Representation). Sun IR is a language- and machine-independent representation that is suitable for global optimization and code generation. The following features of Sun IR facilitate global optimization:

- A language-independent symbol table structure, that explicitly represents storage classes, constants and variables
- Facilities to represent static equivalencing and dynamic aliasing
- A general framework for control flow analysis, data flow analysis, and most advanced global optimization techniques

The following short sections summarize the features of the four front ends that share the common back end.

Sun C

Sun C offers an ANSI C language compliant compiler, as well as an advanced K&R version. With the ANSI C compiler, your programs are fully portable across all ANSI C platforms. The two compilers included in Sun C allow you the flexibility to choose between these different C languages. the Sun C package also provides the following features:

- Function prototyping to ensure better static type checking of programs. Static type checking enables you to find logic errors at compile time, which reduces software development and maintenance time.
- Language enhancements, such as the `const` and `volatile` keywords, help improve program correctness by allowing better control of variables and improving the scope of optimization.
- Multi-byte characters support writing code that can be localized to particular countries.
- `-fsingle` command-line options permit FORTRAN-like floating-point expression evaluation. This option enables developers to write computation-intensive applications in C.

Sun C++

Sun C++ implements the complete C++ language as described in AT&T's C++ *Language System Product Reference Manual*. Sun C++ also incorporates all the functionality of AT&T's latest `cfront` C++ translator. The features of Sun C++ include:

- ANSI C facilities
- Position-independent code generation
- Enhanced `cpp` preprocessor that handles C++ tokens
- A set of classes commonly used in the development of object-oriented programs

Sun FORTRAN

Sun FORTRAN provides an ANSI FORTRAN 77 development system with VAX/VMS™ FORTRAN 4.0 extensions. Sun FORTRAN conforms to the ANSI X3.9-1978 and ISO 1539-1980 FORTRAN standards. In addition, it has been validated by NIST and conforms to FIPS 69-1 BS6832 and MIL-STD 1753. The features of Sun FORTRAN include:

- Extensive VMS compatibility
- Complex expression optimization
- Fast and accurate degree-based transcendental functions
- Support for C preprocessor directives
- DO/ENDDO and DO WHILE statements

Sun has also added several other extensions to the FORTRAN compiler for supercomputer compatibility, including the POINTER datatype and quad-precision floating point.

Sun Pascal

Sun Pascal is an optimizing, feature-rich compiler for Pascal, a widely used, structured language originally designed as an aid for teaching programming. Sun Pascal fully conforms to ISO Level 0 Pascal (equivalent to ANSI/IEEE 770X3.97-1983). Sun Pascal also offers language extensions compatible with many Pascal compilers, specifically those of HP/Apollo DOMAIN® Pascal. Sun Pascal's features include:

- Conformant arrays, as specified in the ISO Level 1 Pascal Standard
- Variable-length string type
- Single- and double-precision IEEE floating-point support
- PUBLIC and PRIVATE declarations
- External C and FORTRAN declarations

After lexical, syntactic and static semantic processing, the remainder of the compilation steps are performed by the back end shared by the C, C++, FORTRAN, and Pascal products.

The machine-independent ("global") optimizer is called `iropt`. It is applied to files and begins by performing automatic inlining, followed by alias analysis, then a series of data flow analyses and transformations are applied to each procedure in the file. For example, data flow analysis could determine that a variable has the same constant value every time control reaches a particular point, and is therefore a candidate for replacement by a constant. The result of the transformations is a modified version of the Sun IR for the program.

Automatic inlining provides many benefits. Obviously modules that have been inlined have no "procedure call overhead." In addition, by moving the body of the module into the caller, many new opportunities for optimization are created. In effect, this provides interprocedural analysis.

The aliaser deals with problems of aliases arising from the presence of multiple names that map to the same memory areas. It is essential to good optimization that the range of possible aliases be determined. Variables that are aliases do not readily lend themselves to optimization. Therefore it is essential to minimize the range of aliases when doing ambitious optimization. In standard FORTRAN, the set of names that may refer to the same location may be determined exactly. This is known as "static aliasing." Languages such as C, C++, Sun's extended FORTRAN, and Pascal introduce an additional challenge called dynamic aliasing. For example, in C, aliases may arise from memory overlaps, array references, use of pointers, etc. Dynamic aliases are defined as aliases that cannot be determined exactly and are therefore given special attention by the aliaser module.

The following additional global optimizations are performed by `iropt`. Definitions of these optimizations can be found in "Appendix A — Optimization Definitions" on page 34.

- Aggregate breaking
- Algebraic address expression reorganization
- Common subexpression elimination
- Complex expression expansion
- Copy and constant propagation
- Dead code elimination
- Induction-variable strength reduction
- Loop-invariant code motion

- Loop unrolling
- Global register allocation
- Tail call optimization

The *inliner* performs inline assembly language expansion. Inline expansion provides a way for the compiler writer or user to specify assembly language code sequences to replace source-language calls. To do this, the compiler writer and/or user provides a collection of "inline template files." The greatest service provided by the assembler inliner is that special code sequences (special supervisor instructions, implementation dependent instructions, etc.) can be accessed without changes to the compilation system. In addition, Sun provides some templates to accelerate performance of the some common library interfaces.

The postpass optimizer performs the following local optimizations. Definitions can be found in "Appendix A — Optimization Definitions" on page 34.

- Constant propagation
- Cross jumping
- Dead code elimination
- Elimination of redundant loads/stores
- Elimination of unnecessary jumps
- Instruction scheduling
- Leaf routine optimization
- Loop inversion
- Register coalescing

The assembler then generates relocatable object code. The linker then:

- Combines separately compiled object files
- Resolves intermodule references
- Searches libraries to satisfy unresolved references

In the case of static linking, the linker combines the relocatable object with other needed relocatable objects (commonly from library files), and produces the executable file. Dynamic linking is a bit more complicated.

Levels of Optimization

The SPARCompilers support several levels of optimization that require various amounts of compilation time and produce correspondingly varying code quality. The default is to do no optimization at all. This is not recommended for any use except debugging, where it is important to minimize compilation time. Each level includes the optimizations of the previous levels. In addition to the "no optimization" level, these are:

`O1`

At this level, only postpass optimization is invoked. Using level `O1` is recommended only if the higher levels of optimization result in excessive compilation time, or running out of swap space.

`O2`

This invokes all the global optimizations, except automatic inlining prior to code generation.

Note – This is the standard optimization level for most modules.

`O3`

This performs the same optimizations as `O2`, but on a wider class of expressions, including references and definitions of external and indirect variables.

`O4`

This level traces, as carefully as it can, what pointers may point to, and makes them candidates for optimization. It also invokes automatic inlining. This level of optimization is recommended for the most computation intensive modules, and not recommended for non-computation intensive modules.

Multiple levels of optimization are provided because aggressive optimization involves trade-offs:

- Compile time vs. execution time
- Memory space vs. execution time

As a rule of thumb, higher levels of optimization increase compile time, decrease execution time and require more memory and disk space to compile programs. Figure 3 shows optimization trade-offs using the SPECmark benchmark test.

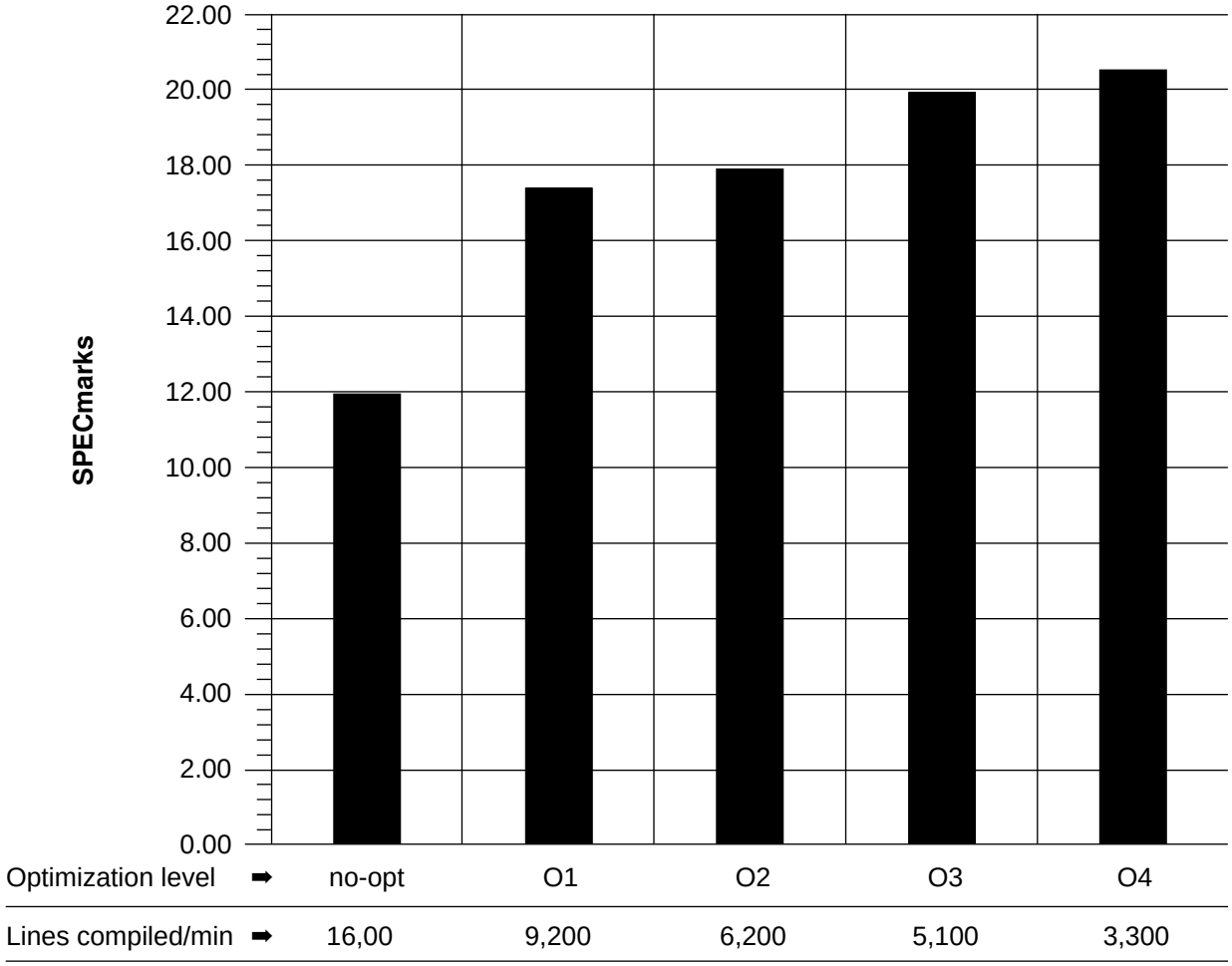


Figure 3 Effect of Optimization Level on Execution Time and Compilation Time. SPECmark tests run on a SPARCstation 2 with SunOS 4.1.4, using Sun FORTRAN 1.4 and Sun C 1.1

For some programs, levels `O3` and `O4` significantly increase compilation time with a small effect on run-time performance beyond that provided by level `O2`. In these cases the developer may choose to compile at level `O2` to avoid the compile-time penalty. In fact, in some cases, by optimizing different procedures at different levels, you can produce overall faster executing code. Use of multiple optimization levels is usually the way to best enhance performance for a large application. Level options can be easily encoded in a Makefile.

Another feature of the back end is automatic back-off of the optimization level. If the compilation of a routine would fail due to lack of sufficient swap space, the optimizer automatically recompiles that routine (only) at the next lower optimization level. When this happens, the user is alerted by means of a warning message.

Selection of optimization flags has been simplified for the common case. Typically, users want a single option, and define their intent as "to generate the best code possible in a reasonable amount of time that runs well on my machines." The `-fast` compiler option is intended to provide this. It provides a convenient way to get near maximum performance with one switch by bundling together several independent options. Any subset of `-fast` attributes can be specified explicitly as indicated below.

The `-fast` option combines:

- Default optimization level: in the absence of an explicit `-O`*n* option following `-fast`, uses `-O2` to obtain the best trade-off between compile and execution time.
- Best choice for compile-time hardware: In the absence of an explicit `-cg`{87,89} on SPARC-based systems, `-fast` generates the fastest code for the hardware of the compile-time machine.
- `-dalign`: assumes double word alignment of double-precision floating-point variables in FORTRAN, unless `-nodalign` is explicitly specified after `-fast`.
- `-fsingle`: for C code, generates single-precision floating-point expression evaluation for single-precision operands.
- `-libmil`: uses the Sun-provided `libm.il` inline expansion templates automatically after any user-specified templates.
- `-fnonstd`: causes hardware traps to be enabled for floating-point overflow, division by zero, and invalid operation exceptions, rather than following the IEEE standard.

`-fast` is an option that provides a feature many Sun users have requested. It picks the most popular options, balancing compilation and execution speeds, assumes that the target machine is identical to the compilation machine and exploits every major compilation feature available. It does not provide the highest level of optitimization (`-fast -O4` accomplishes that), and it does assert that double-precision values are double word aligned; therefore it is not suitable for all programs under all conditions.

Ada

Sun Ada combines a fast, full-featured optimizing compiler with automated program-generation tools for minimal recompilations, library management utilities that enhance compiler performance, and a complete suite of programming tools. The Sun Ada optimizing compiler, based on the Verdix compiler system, is the heart of the Sun Ada language system. Highly tuned for SPARC-based systems, the Sun Ada compiler features exceptionally fast complilation for quick throughput and productive development.

The Sun Ada compiler is constructed of three major components. The front end performs lexical, syntactic and semantic analysis on Ada source code and emits a target independent linear intermediate language (IL). The IL is processed by an optimizer (OPTIM) and then the code generator (CG) produces SPARC object code. The OPTIM optimizing module performs many modern code optimizations, several of which are specific to Ada. The front end also handles some optimizations, such as automatic inlining, that contribute to the speed of the code produced. In addition to optimizations in generated code, a passive task optimization has been introduced that can improve rendezvous times for some common uses of tasks by as much as a factor of eight. Also, the exception tables and look-up algorithms have been optimized to yield fast exception-handling performance.

Figure 4 shows the Sun Ada development environment.

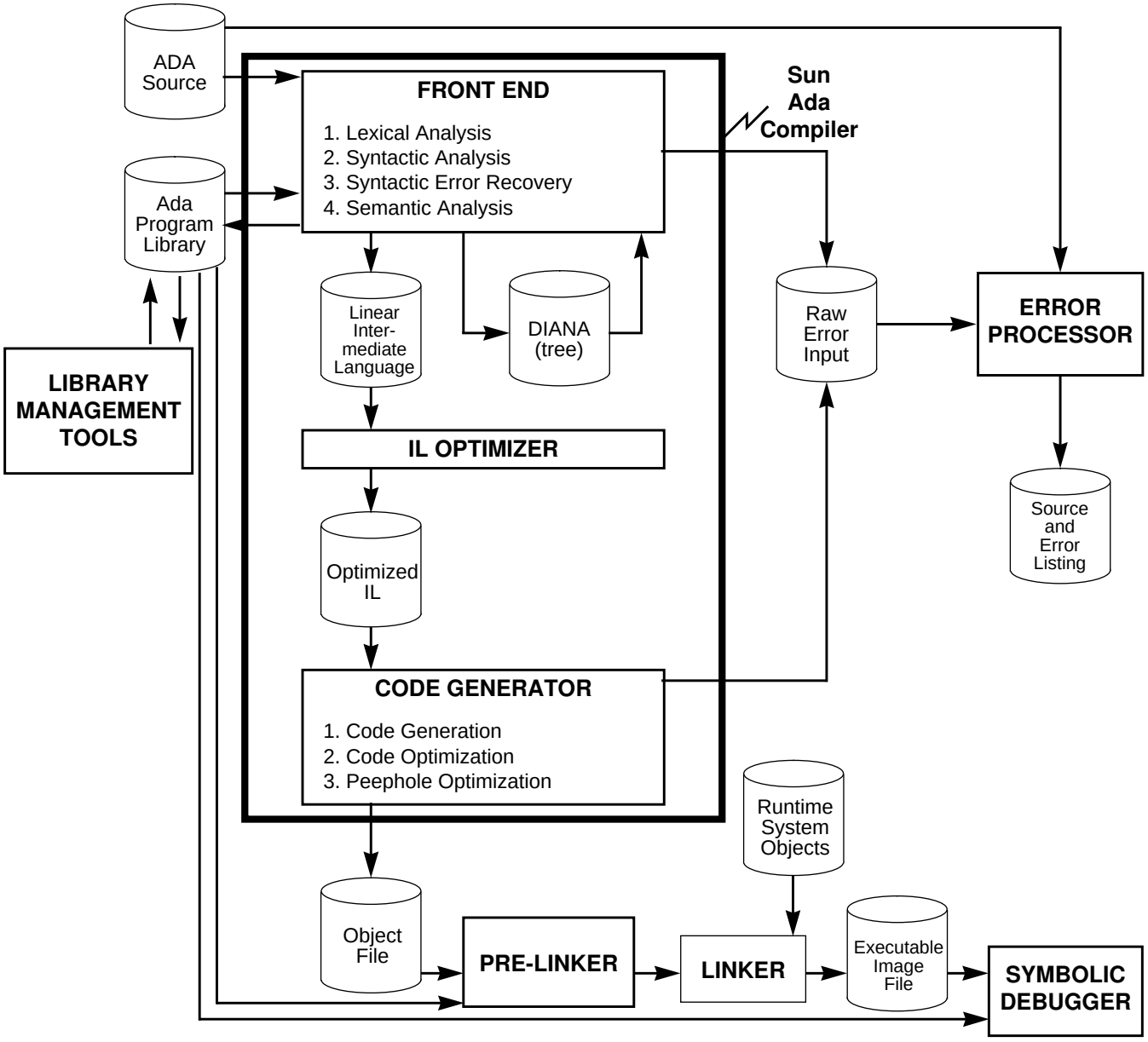


Figure 4 Sun Ada

Optimizations

The following optimizations are performed automatically by the Sun Ada compiler. All three components of the compiler, the front end, the optimizer, and the code generator contribute optimizations. Definitions of these options can be found in "Appendix A — Optimization Definitions" on page 34.

- Algebraic Address Expression Reorganization
- Common Subexpression Elimination
- Constant Folding
- Copy Propagation
- Dead Code Elimination
- Elimination of Redundant Loads/Stores
- Elimination of Unnecessary Jumps
- Induction-Variable Strength Reduction
- Loop-Invariant Code Motion
- Optimized Block Moves
- Range Propagation for Elimination of Constraint Checking
- Register Allocation

OPTIM constructs a flowgraph for each Ada subprogram and then builds a Directed Acyclic Graph (DAG) for each basic block. The optimizer is iterative; it repeatedly applies a set of simple transformations to the flow graph, until no further opportunities for optimization are detected or until a specified iteration limit has been reached. This structure makes the optimizer reliable and easy to understand and maintain. Performance of the optimizer depends more on the complexity of the control flow in the Ada source code than on the number of source lines.

When programming close to the machine level, certain optimizations must be suppressed. Compiler switches offer several levels of optimization and allow optimizations that involve code motion to be suppressed. The pragma `OPTIMIZE_CODE (OFF|ON)` can suppress or re-enable optimization for a specific subprogram or package. The pragma `VOLATILE` (*object_name*) guarantees that loads and stores to the named object will be performed as expected after optimization.

Fast Exceptions

One key Sun Ada design decision is that adding an exception handler to a subprogram (or block) should not slow down the normal execution of that subprogram. In particular, exception handlers do not incur *any* performance penalties unless an exception is raised. This is a highly desirable feature, but it has the drawback of complicating the handling of exceptions. For user code that raises exceptions frequently, this optimization improves performance by computing complex tables at link time that permit high-speed searches for the proper exception handler. As a result of this optimization, Sun Ada customers have the best of both worlds; there is no overhead for normal execution in subprograms that contain exception handlers, and when an exception is raised, exception handling is very fast.

Passive Tasks

The passive task optimization is a textbook example of: "Program semantics exposed by the programming language can be optimized by the compiler." It simply recognizes that a large percentage of all Ada tasks are used exclusively to serialize activities — for example, to serialize access to a data structure. Passive task optimization can be viewed as compiling each accept block into a subprogram and each task entry into a semaphore. The other half of the optimization occurs when compiling a call to the passive task. Instead of generating calls to the run-time system to do a rendezvous, calls are generated to lock the semaphore and do a procedure call. Semaphores and procedure calls are much faster than full-blown rendezvous, especially because they involve only one task (as opposed to two), since the passive task is never made known to the run-time system. On return from the rendezvous, a special resume handler activates one of the tasks suspended on the semaphore, if any.

Passive tasks are a general class of task that includes "monitors." Monitor tasks are extremely important to fast performance in typical Ada real-time situations, and both Sun Ada and the accompanying debugger support it completely.

Remote Compilation

The `-L` option to the compiler and many other Sun Ada tools permits the user to specify the name of the Ada library context in which the compilation is to take place. This means that only one copy of a source file need exist, even if it is shared by projects being developed for different target environments (for example, both the development host and an embedded target).

Large Program Support

Many of the Sun Ada tools have been enhanced to support very large programming projects. For example, the prelinker is designed to quickly compute the elaboration order of large numbers of units.

Interlanguage Calling

The Ada language defines the `INTERFACE` pragma for calling subprograms written in other languages. Sun Ada fully supports the `INTERFACE` pragma for C, C++, FORTRAN, and Pascal. Sun Ada also supports two additional pragmas, `INTERFACE_NAME` and `EXTERNAL_NAME`, which allow Ada to access global data declared in other languages and allow other languages to call Ada subprograms (callbacks). These and other aspects of this topic are covered in detail in the "Interface Programming" section of the Sun Ada Programmer's Guide.

Miscellaneous Optimizations and Pragmas

Ada elaboration order checks are eliminated for packages with static elaboration (for example, those with no dynamic initialization of library level or global variables).

Sun Ada also supports inline subprogram expansion for all types of procedures and functions, including generic and machine-code procedures. Not only does this eliminate call overhead, but it also allows optimizers to work across subprogram boundaries.

Sun Ada shares generic bodies, so multiple instantiations of a generic with similar parameters use the same object code. This saves code space at the expense of execution time. Sun Ada does support unshared instantiation, so actual parameters can be propagated throughout a generic body.

The following are some of the more important pragmas supported by Sun Ada:

pragma `NOT_ELABORATED`
Suppresses the generation of elaboration code for library packages and issues warnings for constructs that require elaboration.

pragma `INLINE_ONLY`
Suppresses generation of a callable version of the subprogram. Otherwise behaves the same as `INLINE`.

pragma `NON_REENTRANT`
Uses a statically allocated parameter block for parameter passing and reduces call overhead. Can only be applied to subprograms nested immediately within a library package.

pragma `NO_IMAGE`
Suppresses the generation of image tables for enumeration types. Use of the IMAGE attribute causes an error message. This does not affect the debugger's ability to display enumeration values symbolically.

Sun Common Lisp

Sun Common Lisp is an implementation of Common Lisp with extensive enhancements to reflect the proposed ANSI Common Lisp standard. Sun Common Lisp is a general-purpose programming language with a rich set of built-in functions for processing both symbolic and numerical data and a wide variety of predefined data types. Sun Common Lisp provides the flexibility that comes from run-time binding of functions and from the fact that Lisp programs can be very naturally processed as Lisp data.

Beyond this, Sun Common Lisp is an interactive programming system that includes:

- An interpreter
- An incremental compiler
- A garbage collector
- Window interfaces
- An object-oriented programming system
- A debugger
- An error-handling facility

Features like the LispView interface for the X Window system, the Multitasking Facility, and the Foreign Function Interface, among others, are major extensions beyond the proposed Common Lisp standard. Many of these development environment features and tools are described in the *SPARCworks Development Environment* white paper. The rest of this section provides an overview of CLOS, the Sun Common Lisp dual compilation system and SPARC support for Lisp.

The Common Lisp Object System (CLOS)

Sun Common Lisp supports the Common Lisp Object System, an object-oriented extension to Common Lisp, that is part of the forthcoming draft ANSI Common Lisp standard. CLOS has its origins in other Lisp-based object-oriented paradigms such as Flavors and CommonLoops. CLOS incorporates the years of experience gained from these models, and has been designed to run on a large array of hardware platforms and operating systems.

Among the fundamental notions of CLOS are classes, instances, generic functions, and methods. Important features of the system are inheritance (including multiple inheritance), method combination, and multi-methods.

For application delivery, Sun Common Lisp offers CLOS extensions that precompile the dispatch code used by generic functions. Although CLOS applications run correctly without a compiler or precompiled dispatch code, they run faster if the dispatch code is precompiled.

Dual Compiler System

Sun Common Lisp has two distinct compilation modes: one that emphasizes compilation speed and one that emphasizes run-time performance.

- The *development mode* compiles the code quickly with few optimizations. Frequent compilations during the development of a Lisp application make compilation speed an important factor in programmer productivity.
- The *production mode* fully optimizes the compiled code for the most efficient run-time performance available. Most users use the production mode of the compiler when they have completed development of a section of code and compile it for the final time.

  The Lisp production mode compiler does constant folding and constant propagation, dead code elimination, and tail call optimization (tail recursion elimination and tail merging). Type declarations are not necessary in Lisp; however, when used they allow the compiler to do further optimizations. With appropriate declarations in the source code, the compiler will
  - Eliminate run-time type checking for arithmetic operations
  - Generate fast code for standard integer computation and for floating-point computation
  - Provide fast array access

  In may cases the compiler will automatically propagate type information to parts of the code that doe not contain declarations.

Development mode is the default, but the user can change the compilation mode by changing the optimization setting of the compiler. In production mode, the user can also specify the amount of run-time error checking, or safety, retained in the compiled code. The development mode inherently retains a high degree of safety.

Users have typically found a three to five times improvement in compilation speed when they use development mode rather than the production mode. Run-time degradation is roughly 50%, depending on the nature of the program. Part of the performance advantage of the development compiler comes from its generating less garbage to be paged or collected.

Optimization Reporting Facility

The user can increase the efficiency of code compiled when using the production mode by providing type declarations that eliminate run-time type checking. In addition, the compiler helps the user optimize code by displaying reports about the optimization attempts that it makes while it compiles code. Optimization reports describe instances where the compiler optimized a section of code and when it did not, but could have, if it had more type information. This useful information allows users to add declarations that improve the performance of an application and reduce the amount of time spent optimizing code.

Foreign Function Interface

Sun Common Lisp provides a Foreign Function interface that allows users to link compiled C, C++, Pascal, and FORTRAN code with Lisp programs and to link Lisp programs into executing C, C++, Pascal, and FORTRAN code. The Foreign Function Interface automatically handles the data type coercions necessary to pass data between Lisp and the foreign code.

Correspondence between Lisp types and a set of low-level foreign data types is predefined, and Sun Common Lisp provides constructs for defining new foreign structure types. Foreign data structures can be accessed in Lisp and passed back and forth between the different languages.

Ephemeral Garbage Collection

The Ephemeral Garbage Collector (EGC) replaces long garbage collection intervals with several shorter intervals that are generally imperceptible to users. Most garbage collections last only a few milliseconds, so that productive development time or execution of critical applications is not interrupted.

When the EGC is on, new Lisp objects are created in a small consing area, which when full, can be collected quickly. Objects from this small ephemeral area that survive the garbage collection process migrate to more long-lived areas of memory where garbage is collected less frequently, resulting in more focused garbage collection of only highly volatile areas. When all ephemeral levels have been filled with objects, the entire system can be collected with the stop-and-copy collector. The EGC can be turned on or off as desired, so that garbage collection does not impact the system at any particular time.

Multitasking (Stack Groups)

Lisp 4.0 includes the capability to run lightweight processes, implemented in stack groups, for multitasking Lisp applications. The Multitasking Facility allows the user to schedule execution of multiple processes running concurrently in the same Lisp environment. The advantage of using the Multitasking Facility is that it allows users to split larger jobs into separate tasks that execute independently.

The Multitasking Facility has its own scheduler that uses state information to stop a process and restart it later without changing the results of its execution. Lisp also provides constructs for handling and scheduling processes that make the implementation of multiprocessing applications both easy and natural. In addition, within Sun Common Lisp the developer can set the priorities of processes to control execution.

SPARC Support for Lisp

In the past, Lisp programs have had a reputation for slow execution compared to other compiled languages. This slowness has been largely due to the high degree of flexibility and the sophisticated error detection and recovery facilities that are the hallmarks of Lisp. For example, each arithmetic operation is generally preceded by one or more instructions that examine the type of each operand and then branch to the appropriate type of operation. Any program that must perform these checks at runtime is at a performance disadvantage when compared to less flexible languages such as C and FORTRAN, which make those decisions during compilation.

One way to make Lisp run faster is to eliminate some of the run-time type checks. The programmer has the option to declare that a given variable will always hold values of a specific type. The compiler can then eliminate the type-checking instructions and produce object code with performance and safety comparable to other programming languages.

The disadvantage of such an approach is that it eliminates some of the flexibility designed into Lisp. The programmer can trade the advantage of flexibility and easier-to-debug programs, for faster execution. On Sun workstations the optimization reports greatly facilitate this step, but it is still a necessary part of the development cycle.

Historically, Lisp machines have been noted for their handling of this problem. In such special-purpose computers, type checking of operands does not require additional instructions. Instead, the check is done as part of the machine instruction either in special hardware or in microcode. Developers could take advantage of all of Lisp's flexibility and still have programs run quickly. This efficiency comes at a price, though: compared to general-purpose workstations, Lisp machines are more expensive, have a less flexible and less open architecture, and are harder to integrate with other systems.

The SPARC architecture used in the Sun-4™ and SPARCsystem product families offers Sun Common Lisp developers many of the more important advantages of the dedicated Lisp machine without the problems associated with a specialized combination of hardware and software. SPARC provides Lisp application programs the potent combination of excellent run-time performance and the ability to detect the most common errors very quickly.

Finally, since a typical Lisp program consists of a large number of small functions, Sun Common Lisp takes advantage of the SPARC register windows. The compiler uses register windows to achieve very fast function calling and argument processing. It is particularly effective in cases that involve a shallow stack of called functions that pass a small number of arguments and programs for which tail call optimization is effective. The speed of this function-calling mechanism can have a major impact on application performance.

Tagged Arithmetic

Lisp provides programmers with a number of interesting opportunities to trade execution time for ease of debugging. In general, for every increase in execution-time performance, there is a decrease in program flexibility and available information for error analysis. As an example, consider the following simple function:

```
(defun adder (first second)
  (+ first second))
```

The programmer has not provided any information as to the type of arguments to be passed to `adder`. Because of this, a test must be done at runtime to identify the kind of addition to be performed. This test adds a sizeable overhead to execution of the function.

If the programmer knows that only `fixnum` (single-precision integer) values will be passed to the function and that the result of the addition will also be a `fixnum`, then it is possible to inform the Lisp compiler of this through type declarations, as shown below:

```
(defun adder (first second)
  (declare (fixnum first second))
  (the fixnum (+ first second)))
```

If the optimization parameters for *speed* and *safety* of generated code are set to maximum and minimum values respectively, then the Lisp compiler is free to bypass the type checking and generate code for `fixnum` addition. However, if either value passed to this faster version of `adder` is not a `fixnum`, then it produces an erroneous result, rather than returning a correct (possibly non-`fixnum`) value or reporting an error. Normally, it is not possible to catch this sort of error without paying a run-time penalty.

SPARCsystem computers offer an elegant solution to this speed vs. safety problem with their tagged arithmetic instructions. These instructions divide the value to be processed into two fields, as shown below:

Special instructions in the SPARC architecture support addition and subtraction between tagged integers. The tag fields are checked at the same time the arithmetic is performed. If the tag fields of both arguments are zero, then they are `fixnum`s and the result is returned. If either tag is nonzero or an arithmetic overflow occurs, then a condition code is set. Optionally, the instruction can cause a trap to occur on a nonzero tag or `fixnum` overflow.

In the fully declared `adder` function above, the tagged add and trap on overflow instruction permits fast execution and still provides excellent error detection. If the arguments and results are as declared, then the generated code wastes no time in error detection. A non-`fixnum` result causes a trap to a routine that allocates and returns an extended-precision integer (a `bignum`). An invalid argument causes a trap to the Lisp debugger.

If declarations are not used, the tagged add instruction uses a condition code to identify its outcome. If both arguments are `fixnum`s, execution time increases over the fully declared function by one conditional branch instruction (in production mode of the compiler). If the tagged add fails, then a generic addition function is invoked that can handle all types of numeric data.

This means that Sun Common Lisp on SPARCsystems can be counted on to deal correctly with any error that results from `fixnum` addition or subtraction without any sacrifice in execution speed. While other computer systems require that programmers risk an incorrect result to get the fastest possible execution, SPARC's tagged arithmetic instructions offer Lisp developers high speed and error detection.

Tag Bits for List Processing

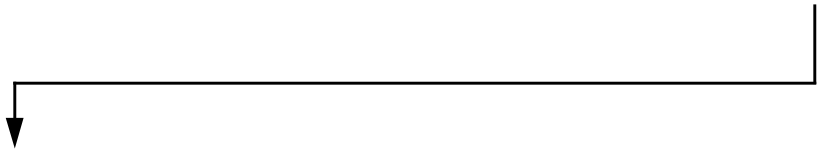
As the name of the language might imply, processing of linked lists is a common activity in Lisp programs. The `car` (find the value of the current list element) and `cdr` (find the next list element) operations are executed frequently. (A list element, in the sense used here, is also known as a cons cell[1].) It is not unusual for a malfunctioning program to attempt to take the `car` or `cdr` of something that is not a valid list element. This is an error that most Lisp systems detect at higher compiler safety settings.

1. The object `nil` is also considered a list element for present purposes; the mechanism described here works correctly for `nil`.

On SPARC-based systems, the misuse of `car` and `cdr` is always detected, even with safety set to 0. Lisp takes advantage of the SPARC architecture's requirement that word-oriented loads and stores require word-aligned addresses, that is, ones that are divisible by four. An invalid address invokes a trap handler that in turn reports the precise cause of the error to the Lisp debugger.

As for `fixnums`, Sun Common Lisp divides pointers into data and tag fields. This time, the data field is the address of the target value without the two low-order bits. In a valid word pointer, these rightmost bits are always zero. In Sun Common Lisp, a list element has a tag value of 1. All other types of Lisp objects have tag values other than 1.

Lisp uses the combined word-address-and-tag value of a list element as its base address. This address, with its tag value of 1, actually points to the second byte of the first word of the list element, as shown:



The address of the `cdr` of a list element is this base value minus 1, and the address of the `car` is the base value plus 3. If the target of the operation is a valid list element, then `car` or `cdr` always produces an address on a word boundary. Taking the `car` or `cdr` of any other type of object always causes a trap, because the calculated address is not divisible by four. Because invalid address detection is an integral part of SPARC address processing, no additional time is necessary to detect this type of error.

This is another case where a compiler safety setting of 0 in Sun Common Lisp offers as high a degree of error detection as a much higher setting on other Lisp systems, without the overhead other systems experience.

Sun COBOL

The Sun COBOL product is based on the MicroFocus™ compiler system. The basic MicroFocus product has been enhanced by Sun Microsystems to provide superior performance on networked file systems. This enhancement was accomplished by replacing the general C-ISAM file handler with the Sun NetISAM™ file handler. The NetISAM file handler gives better network performance for indexed-sequential files across a network. The COBOL section of the *SPARCompiler Benchmark* white paper highlights the dramatic performance improvements attained through the use of this optimized file handler.

Sun COBOL programs can call any procedure using the C parameter-passing mechanism.

The Sun COBOL compiler system was validated in 1990 by the National Institute of Standards and Technology (NIST) as compliant at the high level with ANSI X3.23-1985 "Programming Language COBOL." The system also complies with all other relevant standards, including ISO 1989-1985 and FIPS PUB 21-2, and is XPG3 compliant.

Sun COBOL Compiler and Interpreter

The compiler system has the classic division into front end and back end components that communicate through an intermediate code file. In addition, there is an interpreter/loader utility, `cobrun`, that can execute the intermediate code produced by the front end, and can load the SPARC code produced by the code generator.

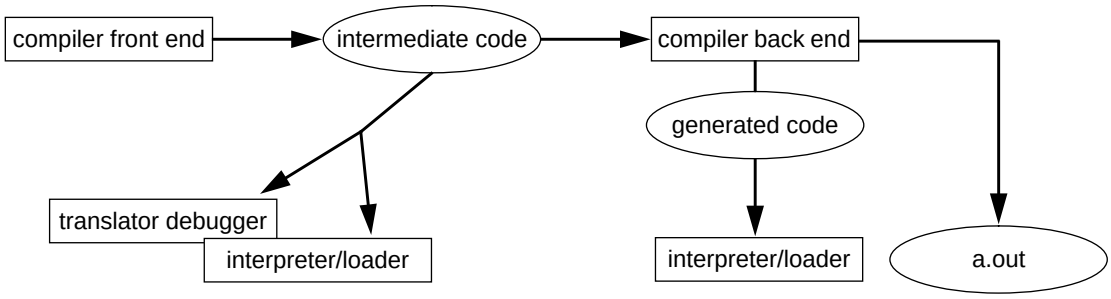
The front end lexically, syntactically and semantically analyzes the source code; the back end is a code generator that outputs SPARC instructions in a form similar to that used in `.o` files, known as "generated code form." This generated code file can be loaded and run using the interpreter/loader utility `cobrun`. The back end can also use `ld` to produce statically linked executable modules in system-standard `a.out` format.

The trade-offs of these different translation alternatives are examined in the next section.

Choosing Between Interpreted, Generated, and Compiled Code

Each of the three translated forms — interpreted, generated, and compiled — has different attributes. Thus, the compiler provides flexibility that allows users to tailor their code development and execution to suit the needs of each specific situation. The code translation alternatives are depicted in Figure 5.

A file that contains interpreted form code can be interpreted on any Micro Focus platform, not only SPARC systems. Interpreted form is particularly convenient for Independent Software Vendors (ISVs) who develop portable applications, or for programmers who want to port an application from a Micro Focus system on other hardware to a Sun workstation.



	interpreted form	**generated code form**	**a.out form**
Compile option:	`cob -a foo.cbl`	`cob -u foo.cbl`	`cob -x foo.cbl`
Produces files:	`foo.int`, `foo.idy`	`foo.gnt`	`foo`, `foo.o`
Attributes:	+ small file + code is statically linked + uses dynamic loading + runs on any MicroFocus platform + can user debugger - slow on CPU-bound code - needs interpreter/loader	+ small file + uses dynamic loading - cannot use debugger + fast on CPU-bound code - needs interpreter/loader	- large file + uses dynamic loading - cannot use debugger + fastest on CPU-bound code + self-contained executable
Uses:	+ debugging + portable applications	+ small, fast executable + benchmarking	+ huge, fast executable

Figure 5 Code Translation Alternatives

Dynamic Loading in Sun COBOL

The "dynamic loading" referred to in this document should be distinguished from the SunOS "dynamic linking" feature. Dynamic loading is a feature of the MicroFocus implementation that loads COBOL modules at runtime, rather than binding them together permanently into an executable at link time. When a UNIX file that contains COBOL source code is compiled to intermediate or generated code form, a `.int` or `.gnt` file is created for each COBOL program within that file. As the programs are called at runtime, the corresponding `.int` and `.gnt` files are dynamically brought into memory by the Sun COBOL Runtime System (RTS) for interpretation or execution. However, dynamic loading does convey an advantage similar to dynamic linking: executables are much smaller than statically linked `a.out` programs because common libraries are accessed at runtime.

There are two recommended uses of dynamic loading of COBOL modules. First, an ISV can construct an application so that its major parts are contained in separate `.int` or `.gnt` files. These parts are then invoked by the main program and dynamically loaded by the RTS as the features they implement are used. When an application is structured in this manner, the ISV can issue updates to customers by supplying only the `.int` or `.gnt` files that require replacement. This structure may significantly reduce maintenance costs of the application.

The second use of dynamic loading is for debugging large applications consisting of many modules. If the code is kept in separate dynamically loadable COBOL modules, individual ones can be recompiled for debugging by simply recompiling those files with the `-a` option of the `cob` command. This procedure debugs only those modules that were recompiled with the `-a` option; this saves significant maintenance and development costs for that application.

Flags, Directives, and Switches for Sun COBOL

There are three methods to communicate information to the compiler and run-time system:

1. Through command-line flags
2. Through compiler options (also known as "directives")
3. Through run-time switches

There are approximately thirty command-line flags. These flags control aspects of the compilation process (such as the form of compilation, the name of the output file, and various linking attributes).

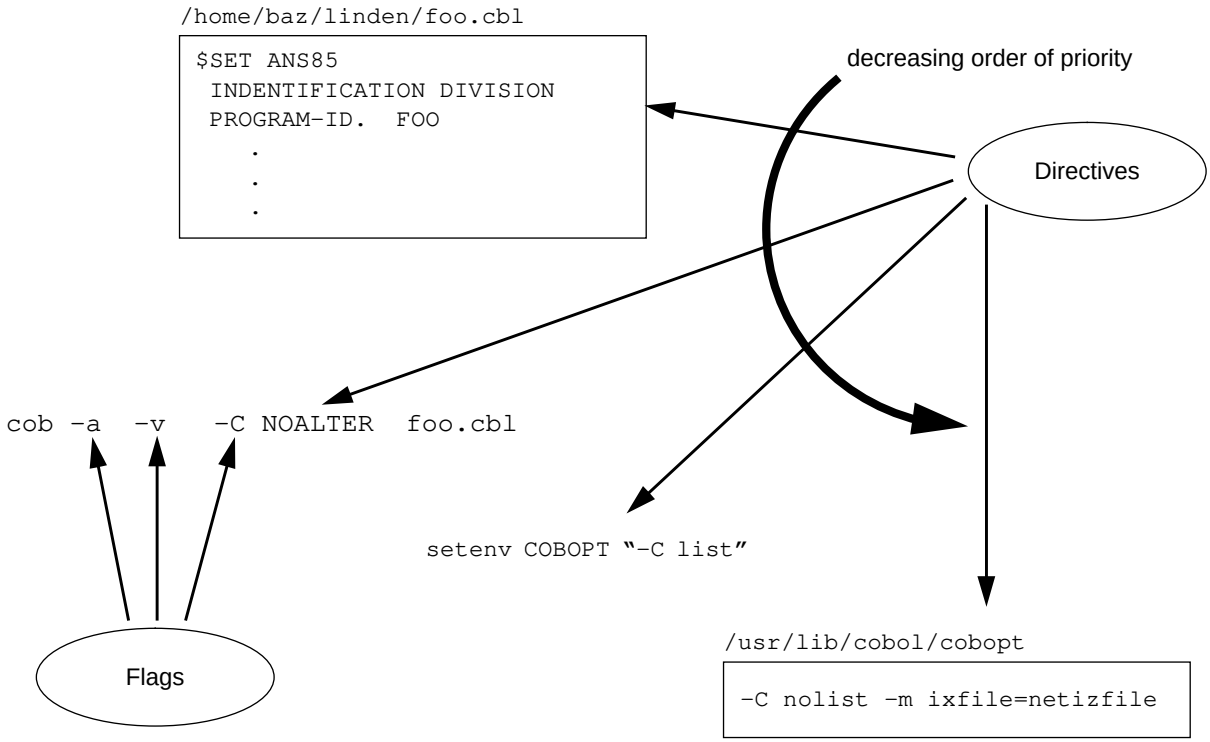


Figure 6 Differences Between Compiler Flags and Directives

There are approximately one hundred compiler options. These options adjust the semantics of the language that the compiler accepts. For example, compiler options are available that recognize COBOL features specific to IBM Microsoft COBOL, to flag features outside a specified dialect, and to change record type defaults to variable length. In addition, there are an additional dozen compiler directives specifically for the code generator. System-wide default compiler directives are set for all programs in the file `/usr/lib/cobol/cobopt`. Directives can also be communicated via the environment variable *COBOPT*, on the command line with the `-C` flag, or embedded in a source file. Finally,

there are approximately twenty switches that affect the run-time behavior of programs. These switches are communicated through the environment variable *COBSW*.

References

SPARCompiler Technology

Muchnick, Steven S. *Optimizing Compilers for the SPARC Architecture*, in Sun Technology, Vol. 1, No. 3, Summer 1988

Also in:

"The Sun Technology Papers", M. Hall and J. Barry (eds), Springer-Verlag, 1990

"Reduced Instruction Set Computers" (Second Ed.), Stallings (ed.), IEEE Computer Society Press, Los Alamitos, CA, 1990

Muchnick, Steven S. *Optimization in the SPARC Compilers*, in Procedures of the Sun Users Group Conference, Atlanta, June 1991

Also to appear in:

Procedures of SUN USER '91, Birmingham, England, September 1991

README (SUG publication), next issue

SPARC Technology

Garner, R. *SPARC: Scalable Processor Architecture*, in "The Sun Technology Papers", M. Hall and J. Barry (eds), Springer-Verlag, 1990

Compiler Technology

Aho, Alfred V., Sethi, R. and Ullman J.D. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986

Appendix A — Optimization Definitions

As used in compilers, *optimization* refers to methods that improve the run-time performance of a compiled program, as compared to one translated by entirely straightforward methods. Optimization algorithms usually operate either on an intermediate code form or on object code.

Most optimizations concentrate on reducing execution time, but a few are specifically directed toward reducing the space a program occupies. Occasionally these goals conflict, so that, for example, a transformation that reduces execution time may increase the size of the object code. This size increase in rarely a problem because it is not usually very significant and, in almost all cases, reducing execution time is much more desirable than reducing the size of an object program.

One of the goals of the SPARCompiler product family is to produce the most highly optimized code possible. This is balanced by providing command-line switches that engage different levels of optimization to suit the needs of the phases of the development cycle. The remainder of this section provides definitions for the optimizations that are mentioned in the body of this report:

- *Aggregate Breaking* — an optimization that allows the individual components of composite objects to be treated as if they were scalars by other optimizations. This optimization is especially well-suited to work with copy propagation, register allocation and inlining. By viewing some structure components as scalars, the optimizer can avoid extraneous memory operations and can enable other optimizations that cannot be used effectively on entire structures.
- *Algebraic Address Expression Reorganization* — systematically transforms address expressions and collects region constants[1] to form simpler expressions from complex ones. A general-purpose transformation engine iteratively applies rules from a transformation grammar until no more simplification can be achieved.
- *Automatic Inlining* — A process whereby the code of a procedure body is placed directly into the body of the caller, in place of the call. This has several benefits:
  - Call overhead is eliminated (this is usually a minor effect)

1. Region constants are expressions that have the same value throughout execution of some segment of a procedure, such as a loop.

- Interprocedural optimizations are exposed (for example, common subexpression elimination, dead code elimination, and register allocation).
- In the current Sun implementation, the caller and the callee must reside in the same text file. A variety of sophisticated heuristics are employed to reduce the probability of adverse performance effects due to code size expansion. Entry points for the inline functions are preserved, so they can be called from functions residing in other files and from the debugger.

- *Common Subexpression Elimination* — saves expression values and reuses them, instead of recomputing them.
- *Complex Expression Expansion* — complex expression expansion works by separating complex expressions into subtrees for the real and imaginary parts of the complex expression. By splitting complex expressions in this manner, it is possible to increase the speed of complex arithmetic because the separate parts of the complex expressions reside in registers during function calls, instead of in memory.
- *Constant Folding* — constant-valued expressions are evaluated by the compiler and the results are inserted into the generated code.
- *Constant Propagation* — a technique that replaces references to variables that are known to contain constant values with the constants themselves. The primary benefit of this optimization is that other optimizations such as constant folding and algebraic simplification can replace runtime computations with ones done at compile time.
- *Copy Propagation* — copy operations that assign a simple value to a variable are of interest to copy propagation if, at runtime, the source of the assignment can be referenced faster than its target. For each such copy, all uses of the target that can be reached by this copy are replaced by the source, if the source is not redefined between the copy operation and its use.
- *Cross Jumping* — a technique that combines identical code found both immediately before a branch instruction and immediately before the branch target. This redundant code can be combined into one sequence.
- *Dead Code Elimination* — information is maintained by the optimizer to track what code is reachable. An expression computation is dead if there is no execution path along which the computation can reach any use of the value it computes. A variable definition is dead if it cannot reach any uses.

- *Detection of Interesting Variables* — a strategy to reduce compilation time by concentrating optimization effort on the parts of programs that are expected to yield the largest improvement for the smallest amount of work. By analyzing the types and patterns of variable references, the optimizer determines which variables are more likely to be candidates for optimization. These "interesting" variables are then targeted for optimization.
- *Elimination of Redundant Loads/Stores* — As an example, the load is redundant in the following case:

```
st %fn, [eq]
ld [eq], %fn
```

- *Elimination of Unnecessary Jumps* — For example:

```
   jmp a  ⎫                          jmp b
   ...    ⎬        becomes           ...
   ...    ⎪                          ...
   ...    ⎭                          ...
a: jmp b  ⎭                       a:jmp b
```

- *Induction-Variable Strength Reduction* — replaces slower operations (for example, multiplications) by faster ones (for example, additions or shifts).
- *Instruction Scheduling* — fine-grained execution parallelism allows several instructions to execute simultaneously, as long as they use distinct functional units. Since SPARCsystems' Floating Point Units (FPUs), and the delay slots following branches and loads, provide such parallelism, the postpass optimizers for these systems rearrange instructions in the generated code to take advantage of this parallelism.
- *Leaf Routine Optimization* — leaf routines (routines that call no others) are comparatively common. If a leaf routine uses few registers and needs no local stack, it can be entered and exited with the minimum possible overhead by omitting the `save` and `restore` instructions and correspondingly adjusting the register numbers used in it. This saves cycles and also reduces the number of register windows employed.
- *Loop-Invariant Code Motion* — finds those computations within a loop that yield the same results for each iteration of the loop and moves them out of the loop.
- *Loop Inversion* — used to convert pre-test loops into post-test loops. This optimization allows loops that have two branches per iteration to be turned into loops that have one branch per iteration.

- *Loop Unrolling* — replaces the body of a loop with several copies of the body, adjusting the loop control code accordingly. This optimization reduces the run-time looping overhead by reducing the number of loop iterations taken. More importantly, increasing the size of the loop body also increases the effectiveness of instruction scheduling.
- *Optimized Block Moves* — tight inline code is generated for block moves when it is known that the source and destination locations do not overlap.
- *Range Propagation for Elimination of Constraint Checking* — once a range check has been performed on an object, the object is tracked so that redundant range checks are eliminated. Related checks can also be eliminated, if, for example, the range of one object is within the range of another. Some null reference checks for pointer types are also eliminated by range propagation.
- *Register Allocation* — decides which objects are worth placing in registers, and which objects can share a register with others within a region of code. For each candidate, the benefit is determined by the number of machine cycles saved by allocating it to a register instead of memory.
- *Register Coalescing* — minimizes the number of registers required to compute a value. Using the fewest registers for one computation ensures that as many as possible are available for use in other computations.
- *Tail Call Optimization* — subroutine calls that are performed immediately before the caller returns are called *tail calls*. By placing a routine's `restore` instruction in the delay slot of the tail call, the called subroutine uses the same register window that its caller used.
- *Tail Recursion Elimination* — converts some self-recursive procedures into iterations. This typically saves register window overflows (on calls) and underflows (on returns) and it saves stack allocation, manipulation and deallocation.

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
FAX 415 969-9131

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Bagshot Manor, Green Lane
Bagshot, Surrey GU19 5NL
England
0276 51440
FAX 0276 51287

Australia: (02) 413 2666
Belgium: 32-2-759 5925
Canada: 416 477-6745
Finland: 358-0-5022700
France: (1) 30 67 50 00
Germany: (0)89-46 00 8-0
Hong Kong: 852 5-8651688
Italy: 039 60551
Japan: (03) 221-7021
Korea: 2-563-8700
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0)8 705 30 00
PRC: 1-8315568
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-7213257
UK: 0276 20444

Europe, Middle East, and Africa,
call European Headquarters:
0276 51440

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales