News

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

FOR IMMEDIATE RELEASE

FOR MORE INFORMATION

Sun Microsystems, Inc.
Erica Vener (415) 336-3566

SUN ENHANCES COMPILERS

New SPARCompiler Optimization Technology Boosts Hardware Performance

MOUNTAIN VIEW, Calif. — April 18, 1989 — Sun Microsystems has enhanced its family of programming language compilers, significantly improving the run-time performance of Sun's SPARC™-based family of workstations and servers. Sun's compilers already are among the most robust in the industry, meaning they generate correct code. There are more than 1,850 SPARCware applications available from Sun Catalyst vendors developed using Sun compilers.

Now called SPARCompilers™, Sun's language products are available for all Sun platforms. They utilize new optimization technology that improves the performance of SPARC systems -- without changes to hardware -- by up to five percent according to the most recent industry benchmarks published by the Systems Performance Evaluation Cooperative (SPEC). In the case of many FORTRAN applications, performance can be boosted 15 percent or more. Besides improved performance, SPARCompilers have been enhanced by new language features, ease-of-use improvements and expanded user documentation. SPARCompiler products are available today for the most recent versions of Sun FORTRAN, Pascal, Modula-2 and C++.

Sun Offers New Unbundled C Compiler

In addition, Sun introduced a new product, Sun C 1.0 -- its first C compiler sold separately from SunOS™, Sun's UNIX® operating system. By unbundling the compiler, Sun can provide more frequent updates and enhancements independent of operating system releases. A version of the C compiler will continue to be bundled and supported with SunOS, but feature enhancements will be made to the unbundled version only.

Improved Tools

Sun C and Sun FORTRAN now include the Sun SourceBrowser, a new window-based search tool. It allows a developer to globally search and locate all occurrences of specific program components through a query or a mouse interface. All SPARCompilers will benefit from an improved version of the dbx debugger, greatly enhanced by the window-based debugging tool, dbxtool. New documentation will also improve the SPARCompiler products. A Numerical Computations Guide and updated dbx/dbxtool manual are now included.

SPARCompilers Superior For SPARC Architecture

SPARCompilers offer several benefits over other language products. They conform to most of the commonly accepted domestic and international standards. They are proven products, with a long track record of successfully compiling thousands of sophisticated user applications. In fact, SPARCompilers were the only language products to correctly compile the SPEC benchmarks the first time, unlike compilers from MIPS, Hewlett-Packard and IBM. And although they run on all Sun systems, SPARCompilers were designed in concert with the SPARC architecture and hardware implementations, to take full advantage of SPARC.

SPARCompilers share many common characteristics and components, such as code generation, SPARC optimization and libraries, that enable them to work well together and give developers a common toolset and development environment. In addition, all Sun languages interact efficiently through common object formats and interlanguage calling capability. Thus programs written in one language can access the libraries of other languages as well as the libraries of Sun system components such as SunOS, OpenWindows™ and graphics. SPARCompilers are all fully integrated with the Network Software Environment (NSE™), Sun's distributed software development environment supporting version control, configuration management and parallel development.

Compiler Support

Sun offers full customer support, training programs and extensive user documentation for its SPARCompilers. The company plans new releases of its SPARCompilers approximately every nine months to provide increased performance and new features. Each SPARCompiler is sold separately and all are available immediately for both SunOS 4.0 and 4.1.

Sun C, C++, FORTRAN and Pascal are priced at U.S. $2,000 quantity one and Modula-2 at U.S. $2,200 quantity one. All prices include media and documentation. Volume and academic discounts are available.

Sun Microsystems, Inc., headquartered in Mountain View, Calif., is a leading worldwide supplier of network-based distributed computing systems, including professional workstations, servers and UNIX operating system and productivity software.

###

For reader inquiries, telephone 1-800-821-4643 outside California. Inside California, call 1-800-821-4642.

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043-1100
415 960-1300

SPARCompiler Facts and New Features

The SPARCompiler Family

Sun's SPARCompiler family of language products today encompasses the most recent versions of Sun C, C++, FORTRAN, Pascal and Modula-2. SPARCompilers conform to most of the commonly accepted domestic and international standards and have a long track record of successfully compiling thousands of sophisticated user applications. Although they run on all Sun systems, they were designed in concert with the SPARC architecture and hardware implementations, so take full advantage of SPARC.

SPARCompilers share many common characteristics and components, such as code generation, SPARC optimization and libraries, that enable them to work well together and give developers a common toolset and development environment. In addition, all Sun languages interact efficiently through common object formats and interlanguage calling capability and are fully integrated with the Network Software Environment (NSE™), Sun's distributed software development environment.

Sun C 1.0

C is the most common language for workstation software development. Traditionally employed for system development in UNIX® environments, it is also widely used for developing all varieties of applications, including database, graphics, windows, communications, business, networking, office automation and word processing. It is the most popular programming language among developers of application software for Sun systems.

New Features:

-- Sun SourceBrowser capability for window-based global searches

-- Sold separately from SunOS™ for more frequent updates

-- Provides support for FORTRAN-like single- and double- precision expressions and user-defined datatypes

-- Improved performance and robustness on Sun386i™; now uses global optimizer

Sun FORTRAN 1.3

FORTRAN continues to be the language of choice for scientific and numerical applications. Sun FORTRAN is an enhanced version of ANSI FORTRAN 77.

New Features:

-- Sun SourceBrowser capablilty for window-based global searches

-- Significant performance improvements over past releases in floating point and complex arithmetic. Seven percent better performance than Sun FORTRAN 1.2; 15 percent better performance than FORTRAN 1.1; even greater performance improvements on applications making heavy use of floating point complex arithmetic

-- Improvements in FORTRAN I/O performance

-- Contains many VAX/VMS FORTRAN 4.0 extensions. This makes it easier for customers to port VAX FORTRAN code to Sun systems. These extensions also allow many FORTRAN customers to use the same FORTRAN source code on both VAX and Sun systems in heterogeneous VAX/Sun environments.

Sun C++ 2.0

C++ is a fairly new but very popular object-oriented programming language. It is an extension of the C language and so provides a natural upgrade path for the C programmer. It is considered to be the language of the future for many C progammers and is being used in many new development projects at Sun.

New Features:

-- Fully integrated with SPARCompiler tools, including the best source-level debugger available for C++

-- The only C++ available that is fully supported on SunOS 4.0 and 4.1

-- Hands-on training is available in Sun's corporate training center and at a variety of training facilities throughout the U.S. Training at customer sites can be arranged.

Sun Pascal 2.0

Pascal was introduced in academia as a language for teaching structured programming. Its popularity has expanded to include PC programmers, who now make up the largest group of Pascal users. Sun Pascal is thus an important language for customers making the transition from PCs to the workstation environment. It is also a popular development language with CAD/CAE vendors.

New Features:

-- Contains many features of DOMAIN Pascal from Apollo, one of the industry's most extended Pascal language products. These features make it easier for customers to port Apollo Pascal code to Sun systems.

-- Fully ISO Level 1 and ANSI compliant

-- Includes both 32 and 64 bit IEEE floating point support for numerical applications

Sun Modula-2 2.2

Sun Modula-2 is a full implementation of this high-level language designed by Niklaus Wirth, the creator of Pascal. Modula-2 was initially designed as a systems programming language to write operating systems and device drivers, but it is also appropriate for large application projects and in education. Today, Modula-2 is particularly popular in Europe.

New Features:

-- Provides native Modula-2 XView bindings for development of graphical user interfaces and graphics applications

-- Includes 64 bit IEEE floating point support for numerical applications as well as 16 bit signed/unsigned integer datatypes

New Programming Tools

Sun SourceBrowser:

Sun SourceBrowser is an interactive window-based tool to aid software developers in the development and maintenance of software systems now bundled with Sun C and FORTRAN. It allows a software developer to find all occurrences of an identifier, string or regular expression through a command line or mouse-driven interface. In addition, SourceBrowser will:

-- Search across all program files, including header files

-- Maintain a list of queries with add, delete and view capabilities

-- Permit users to restrict the search range with user-defined filters and focus

Enhanced dbx/dbxtool:

All SPARCompilers will benefit from an improved version of dbx, a symbolic debugger from Berkeley 4.2BSD UNIX that provides superior execution control and better data display capabilities than standard UNIX debuggers. In addition, Sun has developed dbxtool, a window-based interface to dbx that significantly reduces development time in the debug-edit-compile cycle. The enhanced version of dbx/dbxtool now includes:

-- Support for DOMAIN Pascal extensions

-- Improved handling for FORTRAN array-element syntax

-- A new command that allows user control of debug data volume, simplifying debugging of very large programs

-- Improved diagnostics when debugging corrupted executable files

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043-1100
415 960-1300

Facts About Compilers

A compiler is, most simply, software that translates programs written in the high-level languages (such as C, FORTRAN and Pascal) used by programmers into instructions that can be executed by a computer's processor (such as SPARC or Motorola's 68030 or Intel's 80386). Computers can only understand machine code (also called "binaries") made up of ones and zeroes. People (most particularly, programmers) prefer "high-level" languages that have improved their productivity by being closer to spoken languages like English. Compilers turn the "source code" that comes from programmers into the machine code contained in disks, tape, computer memory, etc.

Compiling is needed in various instances:

- To take a newly written program and turn it into machine code;
- To "recompile" an enhanced/updated version of a program for a particular processor;
- To "port" an existing program to a different processor or operating system.

Many high-level languages exist, since computer users have a wide range of requirements and these languages were developed in different environments -- such as science, education or business -- to meet these requirements. Each language needs its own compiler and each processor needs its own version of that compiler. (You would need, for example, a compiler for Pascal on a PC and a separate compiler for Pascal on a SPARC machine.)

In RISC (reduced instruction set computer) systems, compilers play a much larger role than they do in CISC-based (complex instruction set computer) systems. This is because RISC-based systems have fewer instructions built into the hardware, thus more must be accomplished via software. As a result, a compiler can greatly influence the speed of program execution by optimizing the number and order of software instructions.

Adhering to language standards is important for a compiler so that any program written to those standards can be translated by that compiler. This allows for portability of applications: a standard-based program written for one system can easily be compiled to run on another.

When a compiler is designed in concert with the processor -- as was the case with SPARCompilers and SPARC -- both hardware and software requirements can be taken into consideration at the design time, resulting in the best compilers for that processor. In addition, as the processor evolves, the compilers can evolve along with it.

Sun Microsystems, Inc.

SPARCompiler Optimization Technology Technical White Paper

April 1990

Contents

Chapter 1. Introduction 1-1

Chapter 2. SPARCompiler Overview

2.1 SPARCompiler Structure 2-1
2.2 Global and Postpass Optimizations 2-4
2.3 Levels of Optimization 2-5

Chapter 3. SPARC and SPARCompiler Technology

3.1 Register Windows 3-1
3.2 Delayed Branches and Delayed Loads 3-2
3.3 FPU 3-3
3.4 Pipelining and Parallelism 3-3
3.5 Special Instructions to Support Tagged Data 3-4
3.6 Hardware Interlocks 3-4

Chapter 4. Performance Enhancements 4-1

Chapter 5. Conclusion 5-1

Figures

Figure 2-1. Structure of the Optimizing SPARCompiler Products 2-2
Figure 3-1. Overlapping Register Windows 3-2
Figure 3-2. Sample SPARC Implementation 3-3
Figure 4-1. SPEC Benchmark Performance on SPARCserver 390 4-2
Figure 4-2. FORTRAN Performance 4-3
Figure 4-3. gcc Versus Sun-C 4-3

Chapter 1. Introduction

The purpose of this document is to introduce Sun SPARCompiler™ technology and highlight its interrelationship with the Scalable Processor ARChitecture (SPARC™). It assumes that the reader is familiar with the vocabulary of compilers and computer architecture.

The SPARCompiler system is a suite of products for language compilation on SPARCsystem™, Sun-4™, Sun-3™, and Sun386*i*™ machines, with emphasis on SPARC processor-based systems. SPARCompiler language products are optimized and tuned for improved performance on SPARC processor-based hardware. SPARCompiler products have been developed in conjunction with Sun's SPARC technology and deliver machine-dependent optimization for each SPARC implementation.

Sun's SPARCompiler products include lexical, syntactical, and semantic components — the language "front ends" for the FORTRAN, C, C++, Pascal, and Modula-2 languages. The front ends are integrated with common code generation and optimization modules — the "back ends." The front end for each language closely tracks the corresponding language standards and incorporates useful language extensions introduced by Sun; these extensions provide enhanced functionality and compatibility with other vendors' products.

The SPARC back end utilizes state-of-the-art innovations in compiler technology, particularly in code optimization, code generation, and instruction scheduling. By synthesizing instruction sequences corresponding to Complex Instruction Set Computer's (CISC) complex instructions, compilers for SPARC and other Reduced Instruction Set Computer (RISC) architectures, produce more instructions (approximately 20% more) than comparable CISC machines. These are, however, mostly simple, single-cycle instructions, and easily lend themselves to parallel execution. So, good optimization technology plays a very important role in RISC systems.

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 SPARC hardware implementation, and the code generated by the SPARCompiler products. Because the SPARC ISA and the SPARCompiler designs were developed in concert, the compilers take careful advantage of the architecture in various areas to improve performance: particularly register windows, delayed branches and delayed loads, hardware interlocks, and the Floating Point Unit (FPU), among others. Optimized for SPARC technology, SPARCompiler products play an integral role in the performance of a SPARCsystem.

The System Performance Evaluation Cooperative (SPEC) benchmarking suite is the emerging standard for measuring computer system performance. Continuous technological enhancements in Sun's SPARCompiler back end were demonstrated when the performance of the compilers' latest release was compared with two previous versions, on the same system configuration (the SPARCserver™ 390).

Sun provides compilers for C, C++, FORTRAN-77 with VMS extensions, Pascal, and Modula-2, based on front ends, code generators, and optimizers that are shared wherever possible. In addition, Sun provides Common Lisp (derived from a third-party product), and Independent Software Vendors provide Smalltalk, Prolog, Mainsail, and other languages on Sun platforms through the Catalyst™ Program. All of these products run under the Sun Operating System (SunOS™), derived from the System V and 4.2 BSD versions of the UNIX® operating system.

Chapter 2. SPARCompiler Overview

The SPARCompiler family can be characterized by conformance to standards in the front ends and incorporation of the latest compiler technology advances in the back ends. Focusing all efforts on a single, common back end (per architecture) ensures that performance improvements enhance *all* the compilers.

The following sections describe the SPARCompiler structure. Sun optimization technology is focused on improving SPARC performance. A number of these optimizations are machine-independent, so performance improves on all of Sun's architectures. With the four levels of optimization provided by the SPARCompiler family, users have the flexibility to control compile versus execution time, and memory versus space trade-offs, in the compiled code. The *-fast* option provides a convenient way to obtain maximum performance for the common case.

2.1 SPARCompiler Structure

Figure 2-1 shows how a source program flows through the utilities that transform it into an executable program. The translation steps are usually invisible to the programmer, with the entire sequence of utilities invoked by a single command. The solid arrows describe the path when optimization and in-line code expansion are both enabled. When either is disabled, certain components are skipped, in some cases depending on the source language being compiled.

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

- Support of multiple source languages and multiple target architectures
- Production of very high-quality code without sacrificing execution correctness
- No more than necessary slowdown of compilation for ambitious optimization

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

Preprocessors are programs that manipulate source text, expanding it to a form acceptable by a compiler or assembler. *cpp*, the most widely used preprocessor, 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.

The front end scans and parses the source-language statements constituting a procedure and does static semantic checking. The target of the front end is an intermediate language called Sun Intermediate Representation (IR). Sun IR is a language- and machine-independent representation that is suitable for global optimization and code generation. Features of Sun IR that facilitate global optimization include:

- 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 standard global optimization techniques

The rest of the compilation steps constitute the back end, common across the SPARCompiler family.

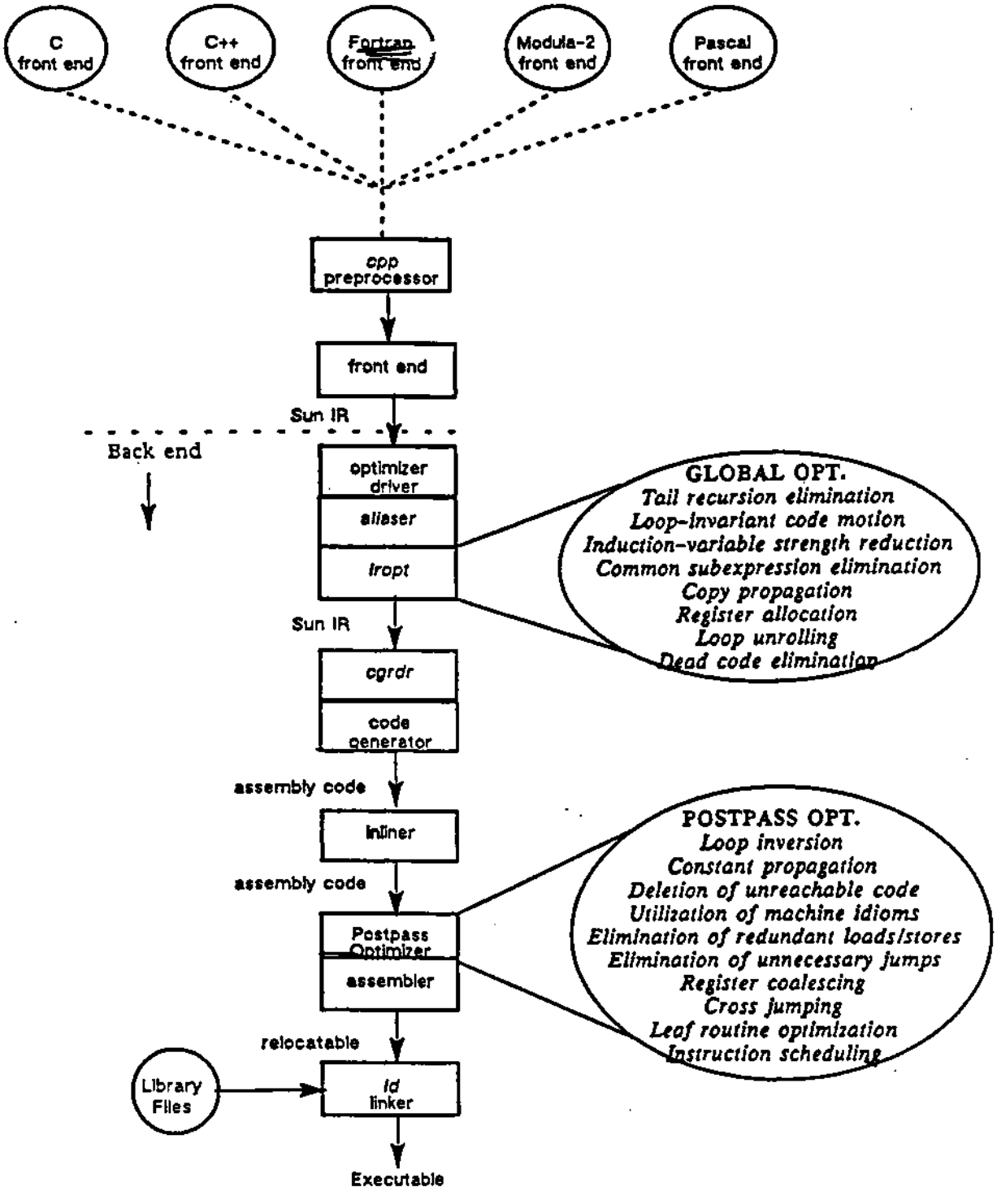

Figure 2.1. Structure of the Optimizing SPARCompiler Products

The optimizer driver reads the Sun IR file, identifies basic blocks[1], and builds lists of successor and predecessor basic blocks for each block, providing enough information to support the extensive control and data flow analysis required to perform global optimization. This information is then passed to the aliaser.

The aliaser deals with problems of aliases arising from the presence of multiple names mapping to the same memory areas. Determining the range of possible aliases is essential to correct optimization. Variables that are aliases do not readily lend themselves to optimization; therefore minimizing the range of aliases is essential to ambitious optimization. In standard FORTRAN, the set of names that may refer to the same location can be determined exactly. This is known as "static aliasing." Languages such as C, Sun FORTRAN with extensions, Modula-2, and Pascal pose an additional problem for the optimizer beyond that posed by standard FORTRAN, namely dynamic aliasing. For example, in C, aliases may arise from memory overlaps, array references, use of pointers, etc. These dynamic aliases, those that cannot be determined exactly, are handled by the aliaser module.

The machine-independent (global) optimizer is called *iropt* and is shared among the compilers for all the architectures. Tailoring it for one architecture or another involves describing the architecture's register model to the global register allocator and tuning a few other components in minor ways. *iropt* is applied to individual procedures and begins by computing further control flow information; most importantly, loops are identified at this point. Then a series of data flow analyses and transformations are applied to the procedure. 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 identified as a candidate for replacement by the constant. The result of the transformations is a modified version of the intermediate representation of the program, which is written back to the IR file. The global optimizations are described in greater detail in section 2.2.

The combination of *cgrdr* and the code generator produces assembly code specific to the machine architecture that is the target of the compilation.

The in-liner performs in-line expansion, which provides a way for the compiler writer or user to specify assembly language code sequences to replace source-language calls. For this, the compiler writer and/or the user provides a collection of "in-line template files." Although such expansions save execution time by replacing procedure calls with in-line code, their greatest benefit comes from introducing opportunities for further optimizations in the postpass optimizer. In-lining achieves many of the benefits of interprocedural register allocation. For example, replacing a procedure call with the in-line expansion of the procedure, makes it a candidate for sharing the same registers as in the original procedure from which it was called. Sun provides highly optimized versions of math routines that may be in-lined in this way.

The postpass optimizer is integrated into the SPARC assembler. It performs machine-dependent optimizations, but is shared across the SPARCompiler family. It does a series of standard machine-dependent optimizations, eliminating jumps to jumps, deleting redundant loads, stores, and unreachable code, inverting loops and several other straight-line code improvements. The postpass optimizations are described in greater detail in section 2.2.

The assembler generates relocatable object code. The linker combines separately compiled files, resolves intermodule references, and searches libraries to satisfy unresolved references. It combines the relocatable object with other needed relocatable objects (commonly from library files), and produces the final executable file.

[1] A basic block is a sequence of consecutive statements that may be entered only at the beginning, and are executed in sequence without any branches, except at the end of the basic block.

2.2 Global and Postpass Optimizations

The global optimizer *iropt* performs a series of data flow analyses and transformations, as detailed in the following sections.

- *Tail recursion elimination* converts some self-recursive procedures into iterations. On SPARC machines, this typically saves register window overflows (on calls) and underflows (on returns), and on all Sun architectures it saves stack allocation, manipulation, and deallocation
- *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
- *Induction-variable strength reduction* replaces slower operations (e.g., multiplications) with faster ones (e.g., additions or shifts). After the strength reduction, if any induction variable is dead, it is deleted by dead code elimination (described shortly)
- *Common subexpression elimination* saves expression values and reuses them, instead of recomputing them
- *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 copy, all local uses of the target that can be reached by this copy are replaced with the sources, if the source is not redefined between the copy operation and the uses
- *Register allocation* decides which objects are worth putting 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
- *Loop unrolling* replaces the body of the loop with several copies of the body, adjusting the loop control code accordingly. This optimization reduces the runtime looping overhead by reducing the number of loop iterations taken. By increasing the size of the loop body, it also increases the opportunity for instruction scheduling
- *Dead code elimination.* Reaching information is maintained by the optimizer to decide what code is reachable. An expression computation is dead if there is no execution path along which the computation can reach it. A variable definition is dead if it cannot reach any uses. The reaching information is updated, at which point the process repeats itself. Unreachable code is eliminated by the postpass optimizer

The postpass optimizer performs the following local optimizations.

- *Loop inversion* converts pre-test loops into post-test loops. This optimization allows loops having two branches per iteration to be replaced with loops having one branch per iteration. The seemingly small benefit gained from removing one instruction is multiplied by the number of times the loop body is executed and can lead to substantial savings in some loops
- *Constant propagation* 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) that perform operations or simplify code at compile-time instead of at runtime can then be done
- *Deletion of unreachable code.* Dead code and unreachable code identified by induction-variable strength reduction and dead code elimination, is deleted at this point
- *Elimination of redundant loads/stores.* This load is redundant:
  st %fn, [eq]
  ld [eq], %fn

- *Elimination of unnecessary jumps*:

jmp a	is optimized to	jmp b
a: jmp b		a: jmp b

- *Register coalescing* minimizes the number of registers required to compute a value. Using the smallest number of registers for one computation ensures that as many as possible are available for use in other computations. This is important because running out of registers necessitates storing some values in memory and subsequently reloading them
- *Cross jumping* combines identical code found just before a branch instruction and the branch target. In this way, the redundant code can be eliminated
- *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 (on SPARC processor-based systems) with the minimum possible overhead by omitting the save and restore instructions and adjusting the register numbers used in it. This saves cycles and also reduces the number of register windows employed
- *Instruction scheduling*. Fine-grained execution parallelism allows several instructions to be executing at once, as long as they use distinct functional units. Sun-3 machines with Floating Point Accelerator (FPA) and SPARC processor-based systems with built-in FPU provide such parallelism, and so the postpass optimizers for these systems rearrange instructions in the code generated for a program to best take advantage of such parallelism

On a SPARC implementation, the instruction scheduler can schedule as many as four types of instructions for execution at each point:

- A branch or load
- An integer instruction other than a branch or load
- Additive floating-point instruction
- Multiplicative floating-point instruction

2.3 Levels of Optimization

SPARCompiler products support several levels of optimization that require various amounts of compilation time. 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 assembly-level optimization is invoked. It is only recommended if the higher levels of optimization result in excessive compilation time or the depletion of swap space.

O2 This invokes global optimizations prior to code generation, including loop optimizations, common subexpression elimination, copy propagation, and automatic register allocation. O2 does not optimize references for external or indirect variables, and pointer variables. This is the generally recommended level for all modules.

O3 This also optimizes references and definitions for external and indirect variables. O3 does not trace the effect of pointer assignments.

O4 This level traces, as assiduously as it can, what pointers may point to, and makes them candidates for optimization. This level of optimization is recommended for the most compute-intensive modules, and not recommended for noncompute intensive modules.

Why are multiple levels of optimization provided? Aggressive optimization implies:

- compile time versus execution time trade-offs
- memory versus space trade-offs

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.

For some programs, O3 and O4 significantly increase compilation time with a small effect on runtime performance beyond that provided by O2. In these cases, developers may choose to compile at O2 to avoid the compile-time penalty. In some cases, by optimizing different procedures at different levels of optimization, developers can get overall faster executing code. Using different levels of optimization may result in the best performance for a large application. This can be easily encoded into a `makefile`.

Compiler directives and feedback from profiling can be used to select the proper settings for optimization switches. *gprof* is a performance analysis utility included in the SunOS that constructs a profile of time spent per routine, subroutine call frequency, and average time spent in each routine per call. It provides extremely valuable information for fine-tuning large applications, by providing information on which routines cause performance hits. Most programs spend 80% of the time in 20% of the code. So by detecting the areas where the program spends most of its time, and only optimizing those, one can save on compile time, with almost the same execution time as when everything is optimized.

Selection of optimization flags has been simplified for the common case. Typically, users want a single option, and would 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 provides for this. It is intended to provide a convenient way to get near maximum performance with one switch. Thus it bundles together several independent options. Any subset of *-fast* attributes can be created explicitly as indicated.

The *-fast* option combines:

- Default optimization level: in the absence of an explicit -On option, uses -O2 to obtain the best trade-off between compile and execution time
- Best compile-time hardware: in the absence of explicit *-f{soft,switch,68881,fpa,fpa+}* on Sun-3 systems or *-cg{87,89}* on SPARC processor-based systems, *-fast* will generate the fastest code for the hardware of the compile-time machine
- *-dalign* (SPARC): assume double alignment of double-precision variables, unless *-nodalign* is explicitly specified
- *-fsingle* (C): generate single-precision expression evaluation for single-precision operands
- *-libmil*: use the appropriate *libm.il* in-line expansion templates automatically after any user-specified templates
- *-fnonstd*: Cause 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 does what many users have requested. It picks the most popular options, balancing compilation and execution speeds, assumes that the target machine is identical to the machine the compilation is taking place for, and exploits every feature available. It does not provide the highest level of optimization (*-fast* -O4 does that), and it asserts that anything is double word aligned if it appears as such, so it is not suitable for all programs under all conditions.

Chapter 3. SPARC and SPARCompiler Technology

RISC architectures, products of the evolution from assembly language to high-level languages, emphasize simplicity and efficiency. CISC architectures were developed to simplify writing applications in assembly language, providing many special-purpose instructions for complex operations. These complex instructions typically take multiple cycles, and complicate the machine implementation. Unlike assembly language programs, the programs compiled by high-level language compilers generally do not use these complex instructions. For example, Sun's C compiler uses only about 30% of the available Motorola 68020 instructions. Studies show that approximately 80% of the computations for a typical program require only 20% of a CISC processor's instruction set. RISC architectures provide a small number of simple, general purpose instructions that can be readily combined by compilers to achieve the same results.

The SPARC design is a new architecture developed by a team of Sun hardware and software engineers. Infrequent operations are performed in software, and only features that yield a net performance gain are implemented in hardware. Performance gains were measured by conducting detailed studies of large high-level language programs. The architecture is licensed by Sun to semiconductor manufacturers for implementation.

As a RISC architecture, the SPARC design achieves high performance by making the things that account for the overwhelming majority of real computing as fast as possible, while not unduly slowing down other operations. Judicious application of this guideline results in an architecture that has a relatively simple set of operations and only a few addressing modes. In the Berkeley RISC II and SOAR tradition, SPARC features include register windows, delayed branches and delayed loads with hardware interlocks, a floating-point coprocessor, and a few special instructions to support tagged data. All instructions are 32 bits long and most take only a single machine cycle to execute. With fixed-format instructions, instruction scheduling for parallel and pipelined execution is made simpler for the compiler.

In the following sections, some of the SPARC features used by the compilers are described. The SPARC and SPARCompiler technology enable Sun to deliver even higher performing systems.

3.1 Register windows

The SPARC design provides three sets of registers: global integer, global floating point, and windowed integer. A unique feature contributing to the SPARC design's high performance is its overlapping register windows. According to the architectural specification, there may be anywhere between 6 and 32 register windows (for windowed integer registers), each window having 24 working registers. An application has access to 24 windowed integer registers at a time, logically divided into three groups:

- Eight *in* registers for communication with the current function's caller
- Eight *local* registers
- Eight *out* registers for communication with called functions

As one function calls the next, the caller's *out* registers become the called function's *in* registers, and a new set of *local* and *out* registers comes into view. The first five arguments to a called function are placed in the caller's *out* registers, with additional arguments stored on the stack. The called function accesses these first arguments through its *in* registers. This scheme simplifies and accelerates function invocation by eliminating the need for the caller to save and restore the state of its registers with every function call. Figure 3-1 shows a diagram of an example SPARC implementation with six register windows. The call and return arrows show the direction in which the register windows turn relative to the Current Window Pointer, during a call and return respectively.

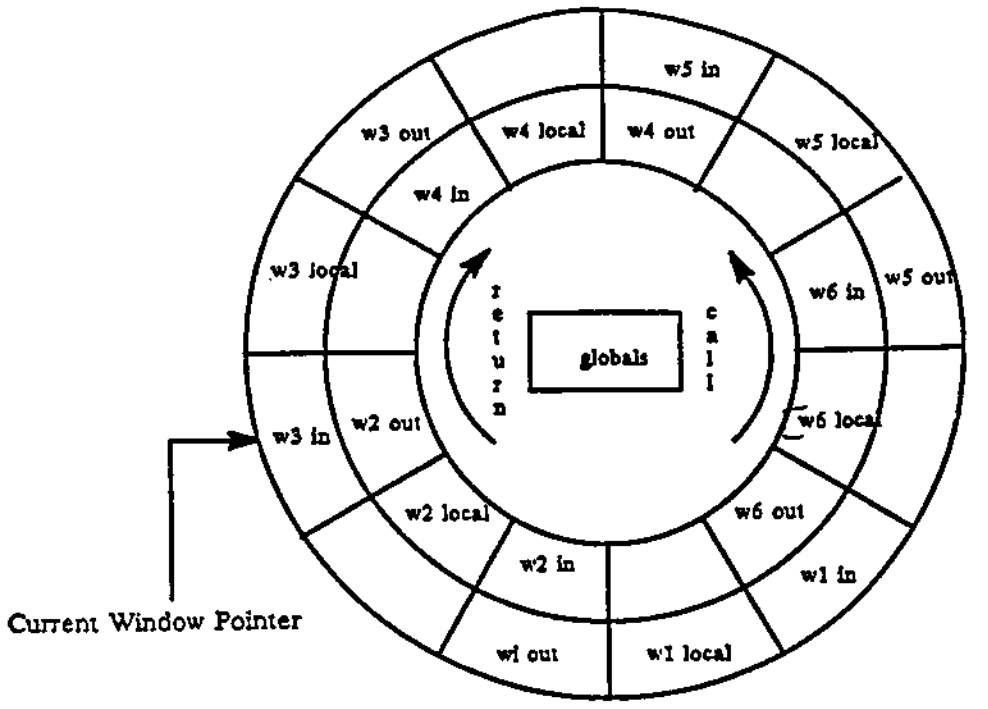

Figure 3-1. Overlapping Register Windows

The effective use of registers is typically among the most important resource allocation issues for a compiler. By utilizing register windows, the SPARC design reduces load/store instructions by about 30%, compared to architectures with no register windows. During a procedure call, the calling procedure's *out* registers become the called procedures *in* registers, saving on loads and stores, and reducing the need for interprocedural register allocation. Without register windows, compilers for other architectures are forced to do more elaborate interprocedural analysis and more complex register allocations at compile time.

3.2 Delayed Branches and Delayed Loads

Instructions that require more than one cycle to execute present compiler writers with a challenge. For example, branch instructions typically require more than one cycle to complete. On a simple machine, it is impossible to overlap the fetching of the instruction following the branch with the execution of the branch; the address of the following instruction is unknown until the branch is complete. SPARC and most other RISC architectures use a delayed branch scheme to allow overlapping branch execution with the fetching and decoding of a subsequent instruction. During the execution of the subsequent instruction, the branch destination is fetched. Sun's SPARCompiler products can fill the branch delay slot with a useful instruction, avoiding a pipeline stall and improving throughput. (A pipeline stall occurs when there is no useful instruction to execute in the pipe, typically filled in by a NOP.)

Load and store instructions may also require multiple cycles because they access memory. By using delayed loads and stores, concurrency is visible to the SPARCompiler products and instructions can be scheduled to occur in parallel with the load or store.

3.3 FPU

The SPARCompiler floating-point environment is the combination of hardware, system software, and software libraries that presents an implementation of the IEEE standard for Binary Floating Point Arithmetic, IEEE standard 754. The kernel provides complete emulation of constructions not implemented in hardware. The math library, *libm*, uses table-driven methods to quickly comput accurate values of elementary transcendental functions.

Like most of Sun's SPARC implementations, the SPARC CPU on the Sun-4/260 is composed of an Integer Unit (IU) and an FPU operating concurrently. (See Figure 3-2.) The IU extracts floating-point operations from the instruction stream and places them in a queue for the FPU. The FPU performs floating-point calculations with a set number of floating-point arithmetic units. The floating point processor, with pipelined floating-point operation capabilities, achieves the high performance needed for numerical applications. The SPARCompiler products optimally fill the instruction stream for the IU and the FPU.

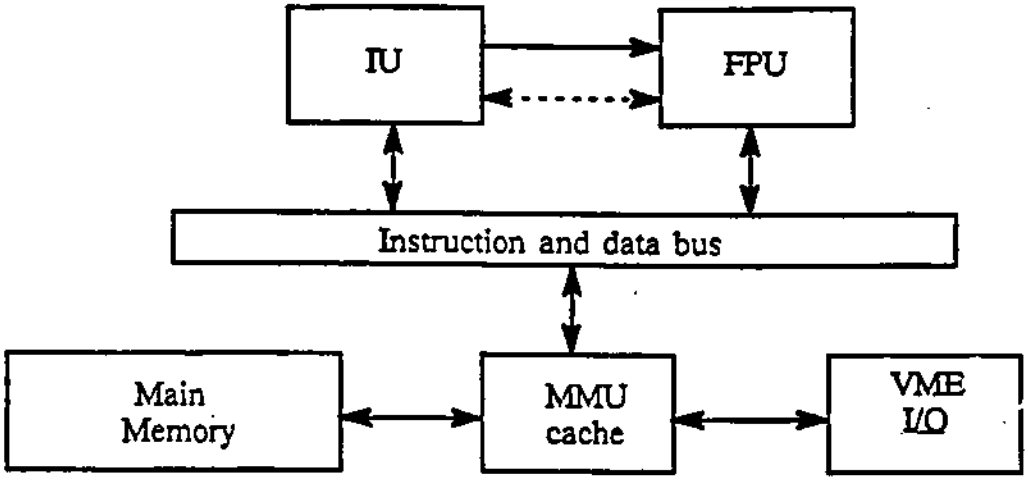

Figure 3-2. Sample SPARC Implementation

3.4 Pipelining and Parallelism

Parallelism allows several instructions to be executed at once, as long as they use distinct functional units. Many characteristics of RISC architectures, such as delayed branches and loads, and an FPU support parallelism. The FPU instructions execute in parallel with CPU instructions, as long as there are no data dependencies between the two units and the FPU instruction queue is not full. For example:

```
fadd x,y,z          The sethi can start on the next cycle even
sethi %hi(loc), %i1 though the fadd has not finished.
```

Although "orthodox" RISC architectures call for all instructions to be one cycle long, floating-point operations typically require much more logic to implement. Forcing all operations to execute in a single cycle would slow down the machine's cycle time, so most RISC implementations have several instructions that take more than one cycle to execute but can be pipelined to execute at the rate of one per cycle.

The SPARC instruction set design allows for the processing of several instructions at the same time. Instruction scheduling and pipelining are critical elements in optimization technology. Instruction scheduling is the art of arranging a sequence of machine instructions so that they complete execution in the minimum amount of time while maintaining correct semantics.

3.5 Special Instructions to Support Tagged Data

SPARC processor-based systems provide instructions that divide the value to be processed into two fields, a 30-bit signed integer and a 2-bit tag. With value types identified by the tags, SPARCsystem tagged arithmetic instructions offer Lisp developers automatic detection of common errors without a performance penalty.

3.6 Hardware Interlocks

Hardware interlocks enable the SPARCompiler products to make optimistic assumptions about data dependencies and do extremely aggressive instruction scheduling, and yet be guaranteed code correct. This is particularly important to ensure binary compatibility across different implementations of the same architecture.

Consider the code fragment:

```
fmult    f1,f2,f3    (4 cycles)
store    f3, [mem]
```

With hardware interlocks, a SPARCsystem will stall until operand f3 is available before executing the store (assuming that no instructions could be found to fill the pipe). In the case of an architecture that does not have hardware interlocks, the compiler would have to fill the instruction pipe with *nops*.

For example, assume that with advancing technology, *fmult* can be executed in two cycles. With the SPARC design, the same binary would now execute the instruction in two cycles. RISC architectures without hardware interlocks will have two choices:

- Be conservative and do pessimistic code generation, maintaining the same pipe length; no gain on new architecture but guaranteed binary compatibility
- Recompile and have multiple binaries for multiple machines; gain cycles but lose binary compatibility. Binaries for faster machines run on slower machines would yield incorrect results

SPARC processor-based systems can be most aggressive and still be correct.

Many SPARC features enable the compilers to provide faster executing and optimized code. The save and restore instructions, executed at the entry and exit of a procedure respectively, make use of register windows to simplify parameter passing. The SPARC register windows permit very fast function calling and argument processing, particularly in cases involving a shallow stack of called functions that pass small numbers of arguments. With delayed branches, loads, and stores, the compiler can fill the delay slots with useful instructions, avoiding a pipeline stall and increasing throughput. The compiler fills the instruction stream for the IU and the FPU, and these execute in parallel whenever possible. Tagged data aids the Common Lisp compiler in delivering higher performance. With hardware interlocks, the compiler does not have to fill the instruction pipeline with *nops*, and the compiler can be more aggressive in optimization. SPARCompiler products exploit these advanced architectural features to maximize software performance.

Chapter 4. Performance Enhancements

Sun's SPARCompiler products are mature and robust. Their continuous development has improved the performance of the compiled code release by release. Sun continues to enhance the SPARCompiler optimization techniques, and the resulting performance of Sun systems, through use of the latest technology and the close coupling of hardware and software development. These advances are verified by increasingly high performance in industry-standard benchmarks.

The SPEC benchmarking suite is a new, evolving standard. It is endorsed by fourteen major competitors whose common goal is "to provide the industry with a realistic yardstick to measure the performance of advanced computer systems" — and to educate the consumer about their products' performance characteristics. Sun is one of the four founding members of this nonprofit organization.

The initial porting effort of the benchmarks onto all member machines was carried out by HP/Apollo, DEC, MIPS, and Sun. In October 1989, SPEC officially announced Release 1.0 of the SPEC benchmark suite containing ten benchmarks. The suite comprises four integer benchmarks (*gcc, espresso, li, eqntott*) and six floating-point benchmarks (*spice2g6, doduc, nasa7, matrix300, fppp, tomcatv*).

These benchmarks were chosen according to the following criteria:

- They are large and run for over a minute on a Sun-4 class machine
- They are based on typical applications
- They are CPU intensive
- The same source will run on all member machines

Future SPEC releases will add more CPU and I/O benchmarks.

The SPEC benchmarks were run on current and previous releases of Sun's compilers. Figure 4-1 shows the SPEC benchmark performance on the SPARCserver 390 for the current and two previous SPARCompiler releases. Overall, the C/FORTRAN performance improved the geometric mean of the SPEC runtime on all the benchmarks: by 7% from f77-1.1/SunOS-4.0cc to f77-1.2/SunOS-4.0cc and by another 5% with the latest release of f77-1.3/C-1.0.

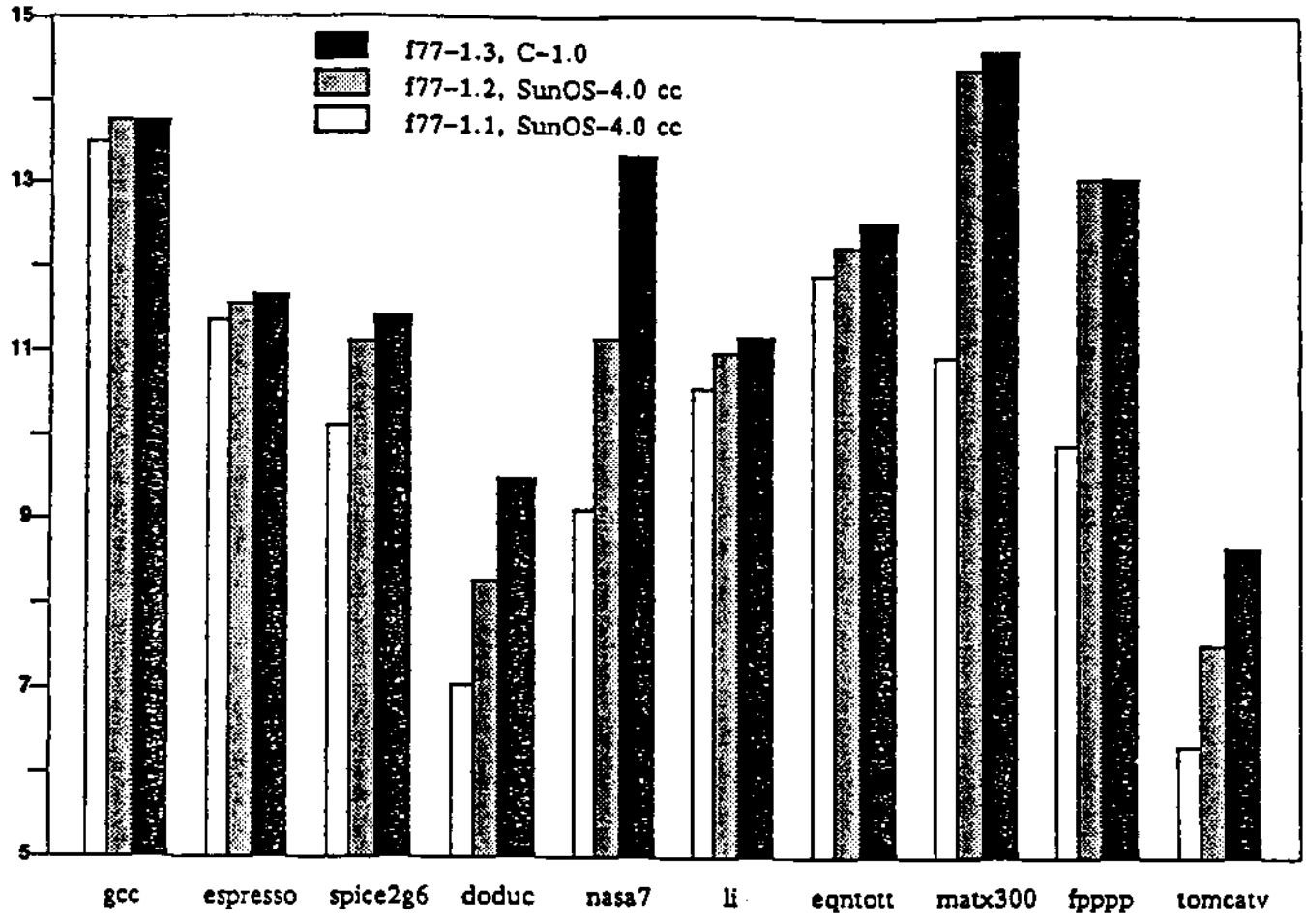

Figure 4-1. SPEC Benchmark Performance on SPARCserver 390

Although the latest SPARCompiler release incorporated many improvements, only one dramatically impacted the SPEC benchmark suite (which, after all, only tests a small part of overall system performance): improved complex arithmetic. Nasa7 improved by 16% for the SPARCserver 390 over the performance of the previous release of the compilers. The acceleration of nasa7 is almost wholly attributable to improvements in complex arithmetic.

Figure 4-2 shows the relative performance of the 1.1, 1.2, and 1.3 releases of the FORTRAN compiler on a SPARCserver 390, as measured on the Release 1.0 of the SPEC benchmark suite (on FORTRAN-only benchmarks).

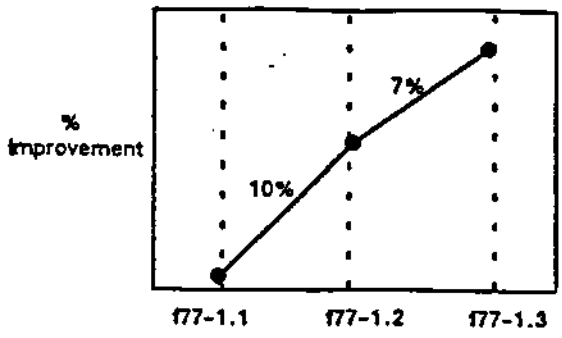

Figure 4-2. FORTRAN Performance

On the C-only benchmarks of the SPEC benchmark suite Release 1.0 measured on the SPARCserver 390, the geometric mean of the SPEC runtimes on Sun C-1.0 was 180 seconds, and on the Free Software Foundation's GNU C (gcc-1.36) was 199 seconds. The C-1.0 compiler performed approximately 10% better than the GNU C compiler.

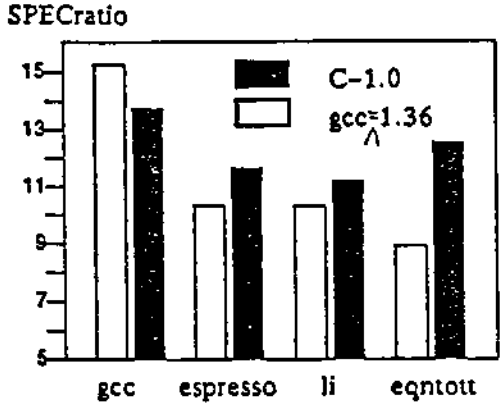

Figure 4-3 gcc versus Sun-C

Chapter 5. Conclusion

The SPARC and SPARCompiler designs have been developed in concert to deliver a superior performing system. This is the result of both better technology in the back end, particularly in code optimization, and the fact that the SPARCompiler products are well integrated with the SPARC design, optimizing system performance.

On a SPARCsystem, the compilers play a significant role in determining performance. Because the languages share the common back end, performance improvements benefit all the languages. The SPARCompiler products have demonstrated improved performance with every new release. On the SPEC benchmarks, the FORTRAN-77 compiler improved by 10% from f77-1.1 to f77-1.2, and again by 7% from f77-1.2 to f77-1.3.

The SPARCompiler family of products is the result of careful evolution of the original Sun compilation system, which itself descended from the original work done at AT&T and UC Berkeley. The SPARCompiler products continue to incorporate new optimization technology without sacrificing robustness, resulting in excellent performance across a wide variety of applications. With its excellent track record, the SPARCompiler family will deliver improved performance with every new release.

Corporate Headquarters
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

European Headquarters
Sun Microsystems Europe, Inc.
Bagshot Manor, Green Lane
Bagshot, Surrey GU19 5NL
England
0276 51440
TLX 859017

Australia: (02) 413 2666
Belgium: 32-2-759 5925
Canada: 416 477-6745
France: (1) 40 94 80 00
Germany: (089) 95094-0
Hong Kong: 852 5-8651688
Italy: (39) 6056337
Japan: (03) 221-7021
Korea: 2-563-8700
New Zealand: (04) 499 2344
Nordic Countries: +46 (0)8 7647810
PRC: 1-8315568
Singapore: 224 3388
Spain: (1) 2532003
Switzerland: (1) 8289555
The Netherlands: 033 501234
Taiwan: 2-7213257
UK: 0276 62111

Europe, Middle East, and Africa,
call European Headquarters:
0276 62111

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

Specifications are subject to change without notice.

Printed in USA 4/90 FE274-0/20K