Floating-Point Programmer's Guide Addendum

Introduction

The *Floating-Point Programmer's Guide for the Sun Workstation*, which describes floating-point programming issues under pre-4.0 versions of SunOS, will be completely rewritten. It now encompasses floating-point programming on both the Sun-4 and the Sun-3 with MC68881 or FPA. This errata summarizes key changes for SunOS 4.0, based upon an early version of the SunOS 4.0 implementation; final results may vary as to speed or correctness. Most of what follows also applies to the special SunOS Sys4-3.2 release.

For more information about the Sun-4, see *The SPARC Architecture Manual.*

This Supplement refers to the C compiler, `cc`, included as part of SunOS 4.0, and to the corresponding FORTRAN compiler, `f77`, sold as a separate product FORTRAN 1.1.

Sun Floating-Point Options

`cc` *Optimization Levels*

Four optimization levels are available for `cc`, namely `-O4`, `-O3`, `-O2`, and `-O1`, described in `cc`(1). `-O1` corresponds to `-O` in SunOS 3.2. Generally, floating-point code may be compiled at `-O4` or `-O3` unless excessively long compilation time results; then `-O2` should provide satisfactory optimization. The default is unoptimized code generation.

FORTRAN Optimization Levels

Three optimization levels are available for `f77`, namely `-O3`, `-O2`, and `-O1`.

Generally, floating-point code may be compiled at `-O3` unless excessively long compilation time results; then `-O2` should provide satisfactory optimization. The default is unoptimized code generation. `-O3` and `-O2` correspond to `-O` and `-P` in SunOS 3.2.

NOTE `foption`*(1) is better than* `-fswitch`

`foption`(1) allows interactive or programmable determination of available floating-point hardware on Sun-3. It's particularly intended for shell scripts that determine at run time which of several executable files to invoke based on available hardware. For instance, such a shell script may select among multiple executables by first switching according to the result of `arch`(1), and then within one CPU architecture, switch according to the FPU architecture with `foption`(1). This executable-level switching is the preferred alternative to the `-fswitch` code-generation option, which imposes a substantial performance penalty on code intended to run on a Sun-3 FPA.

sun microsystems

Rev A, August 1989 Part No: 800-1789-12

Sun-3 Multiple Libraries and Inline Expansion Templates

On a Sun-3, the multiple directories

```
/usr/lib/{ffpa,f68881,fswitch,fsoft}
```

contain versions of `libm.a` and `libm.il` optimized for the indicated floating-point code-generation option. The correct version of `libm.a` is selected automatically when the compilation option is included on the link-step line:

```
cc -ffpa anyc.o -lm
f77 -ffpa anyf.o
```

will both link automatically with `/usr/lib/ffpa/libm.a`.

Inline expansion templates are not used automatically by the compilers; they must be explicitly specified by the programmer when needed. Inline expansion templates are especially recommended for use when compiling any FORTRAN program using complex or doublecomplex variables, and may provide significant performance increases in some other FORTRAN and C programs as well.

The names of the Sun-supplied inline expansion template files have changed since SunOS 3.2. The correct template file is `libm.il` in the directory corresponding to the floating-point code generation option:

```
cc -O4 -ffpa anyc.c /usr/lib/ffpa/libm.il
f77 -O3 -f68881 anyf.f /usr/lib/f68881/libm.il
```

The correct template file must be specified by the programmer.

NOTE *For maximum performance, link* `-lm` *prior to* `-lF77` *on a Sun-3.* FORTRAN 1.1 contains only one version of `libF77.a`, compiled for default floating point code generation (`-fsoft` for Sun-3). Certain FORTRAN library routines are also contained in `libm`, optimized for specific code generation options. Thus in the following:

```
f77 -ffpa any.o
f77 -ffpa any.o -lm
```

the first link will search libraries in the order `-lF77 -lI77 -lU77 -lm -lc`, while the second will search in the order `-lm -lF77 -lI77 -lU77 -lm -lc`. Any Fortran-required routines contained both in `libm` and `libF77` will be linked from the `-fsoft`-compiled library `/usr/lib/libF77.a` in the first case, and from the `-ffpa`-compiled library `/usr/lib/ffpa/libm.a` in the second case.

Constant Expression Evaluation

Some early versions of the manual were incorrect about constant expression evaluation. `cc` and `f77` evaluate expressions involving only integer constants at compile time. `cc` also evaluates expressions involving floating-point constants at compile time. `f77`, however, evaluates expressions involving floating-point constants at run time whenever possible; exceptions arise in cases like

```
parameter(f=1.0+epsilon)
```

which must be evaluated at compile time.

Suppressing Mixed-Code-Generation Warnings

Sun-3 compilers attempt to prevent accidentally mixing `-f68881` and `-ffpa` modules by means of a low-technology trick: such modules include an unsatisfied reference to `f68881_used` and `ffpa_used` respectively. These entry points are only defined in `/lib/Mcrt1.o` and `/lib/Wcrt1.o` respectively; `ld` chooses one of these according to the `-f...` option it gets. Modules compiled with the wrong option cause the following error messages:

```
Undefined:
ffpa_used
```

or

```
Undefined:
f68881_used
```

This safety method no longer works in SunOS 4.0 when programs are linked dynamically (the default); the unsatisfied reference, which is never actually used, may remain unresolved indefinitely. Consequently the sequence

```
cc -c -ffpa any.c
cc -f68881 any.o
```

will link but will not execute correctly because the Sun-3 FPA initialization code normally included in the link step has been omitted.

Static linking, the default in SunOS 3.2, is invoked with `-Bstatic` in 4.0, and will detect such errors. Occasionally sophisticated users will have valid reasons to bypass such error checking. This may easily be done by including assembly-language modules that define the unsatisfied externals. Users doing so are responsible for correctly initializing any floating-point devices they do use.

Sun-4 Considerations

Release 4.0 is the first SunOS release to support both Sun-3 and Sun-4.

No `-f...` Code Generation Options

Unlike Sun-3, there is only one Sun-4 floating-point architecture, SPARC, so no `-f...` option is needed to specify it. But `-fsingle` and `-fsingle2` *are* available on Sun-4 just as on Sun-3 floating-point architectures.

Some SPARC Instructions Implemented in Software

The SPARC architecture specifies certain instructions that are not implemented in the Sun-4/260 or 280 hardware. These instructions are therefore not generated by the Sun compilers. However the instructions are recognized by the Sun-4

assembler and implemented (slowly) by software in the Sun-4 kernel under SunOS 4.0 (but not Sys4-3.2), so that they may be invoked by assembly-language coding if desired. These instructions include:

- `fsqrt[sdx]`
- all extended-precision instructions

Other instructions were listed in early editions of the SPARC manual, but later deleted from the architecture, without ever being implemented. These include:

- `fint[sdx]`
- `fintrz[sdx]`
- `fclass[sdx]`
- `fexpo[sdx]`
- `fscale[sdx]`
- `frem[sdx]`
- `fquot[sdx]`
- `f[sdx]toir`

SPARC Floating-Point Controller

The SPARC CPU board contains two large Fujitsu gate arrays, the SPARC CPU and the floating-point controller (FPC) next to the Weitek 1164/1165. All such prototype FPC's in early Sun-4's should have been replaced by Sun Customer Service with a "FAB-4" FPC. The suffix is at the end of the part number on the first line of the label on the FPC. If the FPC is examined and a prototype FPC found, notify Sun Customer Service. Prototype FPC numbers include:

- MB610303 MB610303A MB610303B MB86910
- FAB-4 FPC's may be numbered
- MB610303C or MB86910A

Don't Forget to `MAKEDEV` fpa!

New I/O devices are not usable on a Sun until they have entries in `/dev`. In most cases these entries are not built into the distributed Sun software and must be done manually, once, by system administrators.

The Sun-3 FPA (and the Sun-2 Sky FFP) are treated as I/O devices by `vmunix` and must have `/dev` entries made and then the system rebooted in order to get the microcode loaded. The `/dev` entries must be remade every time a major software installation occurs. The simple but often-overlooked procedure is to become root and then:

```
# cd /dev
# MAKEDEV fpa
# fastboot
```

FPA and NFS

As a rule, computational servers and file servers don't mix very well; servers should be allocated to one purpose or the other. In particular, Sun-3's with a heavily-used FPA sometimes provide very poor NFS response. The VMEbus timeout setting of the FPA seems to have a significant effect; changing it from 5 microseconds to 4 has helped several systems. If NFS response is a problem in a system with a heavily-used FPA, try the following; if it does not help, undo the change you made in case it was incorrect.

On the Sun-3 FPA, change the settings on the dip switch near the VMEbus from:

```
X=closed  O=open

1 2 3 4 5 6 7 8
O X O X O X X X
```

to:

```
1 2 3 4 5 6 7 8
X X O X O X X X
```

Floating-Point Numerics

Floating-Point Types

Despite extensive documentation available in the IEEE Floating-Point Standard, the MC68881 manual, and the SPARC definition, many questions arise about the details of IEEE floating-point format. For machine-independent coding, the following suffices:

IEEE Format	Exponent Bits	Significand Bits	Equivalent Decimal Precision
IEEE Single: C float Fortran REAL	8	24	6-9
IEEE Double: C double Fortran DOUBLEPRECISION	11	53	15-17
IEEE Double-extended:	15	64	18-21

IEEE Double-extended is the type of the MC68881's floating-point data registers but is not directly available in Sun's high-level programming languages.

"Equivalent decimal precision" is defined as a range; the smaller number is the greatest number of significant decimal digits that is never more precise than the binary type; the larger number is the least number of significant decimal digits that is never less precise than the binary type.

The ranges of floating-point types can be conveniently determined with a simple program like:

```
real r_min_subnormal,r_max_subnormal
r_min_normal,r_max_normal
doubleprecision d_min_subnormal,d_max_subnormal
d_min_normal,d_max_normal

print *, r_min_subnormal()
print *, r_max_subnormal()
print *, r_min_normal()
print *, r_max_normal()
print *, d_min_subnormal()
print *, d_max_subnormal()
print *, d_min_normal()
print *, d_max_normal()
end
```

whose output is

```
    1.40129846E-45
    1.17549421E-38
    1.17549435E-38
    3.40282347E+38
    4.9406564584124654-324
    2.2250738585072009-308
    2.2250738585072014-308
    1.7976931348623157+308
```

New Include Files

Floating-point definitions are now contained in three files. `<sys/ieeefp.h>` contains certain definitions of IEEE floating point required in the kernel. `<floatingpoint.h>` contains definitions required and functions implemented in `libc.a`, including the necessary definitions for correctly-rounded base conversion. `<math.h>` defines the functions implemented in each version of the expanded `libm.a`. `<math.h>` includes `<floatingpoint.h>`, which in turn includes `<sys/ieeefp.h>`. See `floatingpoint`(3) and `intro(3M)`.

New* `libm` *Functions

The mathematical function library, `libm.a`, has been substantially respecified and reimplemented. All (3M) man pages should be reviewed. Some of the changes affect `atan2`(0,0), `pow`(x,0), `hypot` (∞, `x`), and so on.

FORTRAN `libm` *Functions*

All relevant `libm` functions are provided in double-precision and single-precision versions designed to be called from FORTRAN. Thus the C double and single-precision codes

```
#include <math.h>

double x, y;
y = asinh(x) ;

float x, y;
FLOATFUNCTIONTYPE t;
t =  r_asinh_(&x);
ASSIGNFLOAT(y, t);
```

correspond to the FORTRAN codes

```
doubleprecision x, y, d_asinh()
y = d_asinh(x)

real x, y, r_asinh()
y = r_asinh(x)
```

as described in `single_precision`(3M), `libm_single`(3F), and `libm_double`(3F).

`abrupt_underflow` *Mode*

Unlike the asynchronous MC68881, synchronous high-performance floating-point chips, such as the Weitek 1164/1165, used in the Sun-3 FPA and in the Sun-4, are unable to efficiently handle subnormal operands or results in the manner intended by the IEEE floating-point standard. Consequently, software emulation is required to remedy the deficiencies of the hardware by "recomputation" of the correct result from the operands. This is occasionally observed to cause numerical programs to suffer extremely poor performance, consuming a very large amount of system time relative to user time. Part of the system time is due to kernel overhead, and part due to the time required to do the recomputation.

There are three common cases in which such recomputation adversely affects performance.

1) *Underflowed results* on multiplication or division, which are not directed to be trapped by `ieee_handler`(3M), are recomputed to determine the correct subnormal or zero result.
2) *Subnormal operands*, usually the result of previous underflows, are recomputed to determine the correct result.
3) *Exponentials of large negative arguments*, such as double-precision`exp(x)` for `x` < -709, are recomputed to determine the correct subnormal or zero result.

SunOS 4.0 provides a uniform way to obtain `abrupt_underflow` mode treatment of the 1164/1165; the C and FORTRAN calls

```
abrupt_underflow_();

call abrupt_underflow()
```

enable `abrupt_underflow` mode treatment on Sun-3 FPA and Sun-4, which partially corresponds to the "FAST" hardware mode bit of the 1164/1165. `abrupt_underflow` has no effect with other Sun-3 floating-point options. Its effects in the three common cases are currently as follows:

1) *Underflowed results:* `abrupt_underflow` does not affect this case in SunOS 4.0.

2) *Subnormal operands:* On Sun-3 FPA and Sun-4, causes`abrupt_underflow`mode to be treated as zero without causing any exception.

3) *Exponentials of large negative arguments:* On Sun-3 FPA and Sun-4, abrupt_underflow mode causes exponentials, that would normally underflow to subnormal or zero results, to return zero without causing any exception.

In each case, the `abrupt_underflow` mode may only improve performance when the other modes affecting rounding direction and precision have their default values.

`abrupt_underflow` mode does not conform to the IEEE Standard, and IEEE-based exception handling should not be relied upon when running in `abrupt_underflow`mode. To return to Standard-conforming behavior in a program, use the following calls:

```
gradual_underflow();

call gradual_underflow()
```

NOTE *Avoid the obsolete* `fpamode()`.

In general, the Weitek 1164/5 FAST mode is intended to bypass normal IEEE exception handling. Thus not only may the numerical results differ from normal IEEE results, but neither IEEE exception reporting, through `ieee_flags()`, nor IEEE trapping, through `ieee_handler()` and `SIGFPE`, should be relied upon. The `abrupt_underflow` function call is thus named for FAST mode's principal application, although FAST mode could affect other exceptions besides underflow, depending on the floating-point chips and their controllers in a particular implementation.

Porting Applications from non-IEEE Systems

Porting an application from one computer to another always produces interesting side effects, particularly when floating point is involved. The most common porting situation involving Suns is converting applications written for DEC's VAX hardware, VMS operating system, and extended VMS FORTRAN compiler.

Sun FORTRAN 1.1 now supports most VMS FORTRAN extensions, so that use of non-standard FORTRAN is no longer a serious impediment. However, running programs developed under non-IEEE arithmetic for the first time on an IEEE system often reveals previously unsuspected properties of the programs. Often these have to do with exceptions, particularly underflow. VMS FORTRAN programs typically expect that underflow will be treated by underflowing to zero without generating any exception, and that overflow and division by zero will generate a `SIGFPE` that is fatal unless a `SIGFPE` handler has been established. `SIGFPE` handlers often assume that the only cause of `SIGFPE` is an overflow or division by zero. Therefore:

- Programs that underflow frequently and therefore perform poorly may sometimes benefit from `abrupt_underflow()` as described above. Changing from gradual to abrupt underflow may have an adverse effect on accuracy, however; previously abrupt underflow may have been degrading results silently on the non-IEEE system.
- Programs that depend on halting in the event of overflow or division by zero should use `ieee_handler()` to obtain such treatment since the IEEE Standard requires default non-stop exception handling.
- Programs that install their own `SIGFPE` handlers must be rewritten to work properly with the Sun-3 FPA, to recognize and treat appropriately the particular `SIGFPE` code, `FPE_FPA_ERROR`, as described in the 3.2 floating-point manual.

That floating-point exceptions are occurring in the normal mode might be inferred from very slow performance with much system time relative to user time, or from messages produced by `ieee_retrospective()`.

Trigonometric Argument Reduction Mode

Trigonometric functions for radian arguments outside the range $[-\pi/4,\pi/4]$ are usually computed by "reducing" the argument to the indicated range by subtracting integral multiples of $\pi/2$.

Since π is not a machine-representable number, it must be somehow approximated; the error in the final computed trigonometric function depends on the rounding errors in argument reduction with an approximate π as well as the rounding and approximation errors in computing the trigonometric function of the reduced argument. Even for fairly small arguments, the relative error in the final result may be dominated by the argument reduction error, while even for fairly large arguments, the error due to argument reduction may be no worse than the other errors. See the March 1981 issue of the IEEE's **Computer** magazine, page 71.

There is a widespread misapprehension that trigonometric functions of all large arguments are inherently inaccurate, and all small arguments relatively accurate, based on the simple observation that large enough machine-representable numbers are separated by a distance greater than π. However there is no inherent boundary at which computed trigonometric function values suddenly become bad, nor are the "inaccurate" function values useless. Provided that the argument reduction be done consistently, the fact that the argument reduction is performed with an approximation to π is practically undetectable, since all essential

identities and relationships are as well preserved for large arguments as small.

There are several consistent ways to perform trigonometric argument reduction; SunOS 4.0 provides three. Perhaps most satisfying to mathematicians is to generate an approximation to π of such precision that the roundoff due to argument reduction is never worse than any other roundoff in the final answer. That's as good as having π to infinite precision. For IEEE double precision, this is equivalent to an approximation to π of over one thousand bits of accuracy, so this method is slowest.

Most satisfying to Sun-3 users is to perform argument reduction with the 66-significant-bit approximation to π used in the MC68881 hardware and mimicked in the Sun-3 FPA hardware. This is most efficient on Sun-3's.

Some users prefer that trigonometric argument reductions be performed with an approximation to π representable as an ordinary floating-point variable; for those users, the nearest 53-significant-bit double-precision approximation to π is available. Call that approximation P: it has the property that computed `sin(P)` == 0 just as the correct `sin(`π`)` == 0; furthermore `tan(P)` == ∞. 53-bit reduction is the most efficient on Sun-4's.

The trigonometric argument reduction mode is selected by assigning to a global C variable `fp_pi`. Its allowed values are described in <`math.h`>:

```
enum fp_pi_type {
     fp_pi_infinite = 0,      /* Infinite-precision approximation to pi. */
     fp_pi_66 = 1,            /* 66-bit approximation to pi. */
     fp_pi_53 = 2}            /* 53-bit approximation to pi. */

extern enum fp_pi_type fp_pi;
```

`fp_pi` is initialized to `fp_pi_66` in order to produce the same results as for programs previously run on Sun-3's. `fp_pi` is directly available to C programs; from FORTRAN it is necessary to call a short C program like this:

```
call set_pi_53()
...

#include <math.h>

void
set_pi_53_()
{
      fp_pi = fp_pi_53;
}
```

The relative performance of infinite, 66-bit, and 53-bit argument reduction varies considerably depending on available hardware and on the magnitudes of the arguments. When most trigonometric arguments are typical ones of magnitude < 10, then the choice of argument reduction constant usually has insignificant performance impact.

NOTE *Using the* `f68881/libm.il` *or* `ffpa/libm.il` *inline expansion template files will always cause 66-bit argument reduction to occur regardless of the setting of* `fp_pi`.

IEEE-Standard Conformance

Correctly-Rounded Base Conversion

C and FORTRAN input and output of decimal floating-point numbers is now correctly rounded to or from the internal binary representation. For extreme exponents, "correctly rounded" is more exacting than the IEEE minimal requirement. Correctly-rounded base conversion is obtained through the usual C functions `strtod`(3), `scanf`(3), and `printf`(3), and through the usual FORTRAN input and output. These functions now can read, as well as write, ASCII representations such as "inf," "infinity," and "NaN."

Finer control, including access to rounding modes and exception flags, can be obtained by use of `string_to_decimal`(3), `decimal_to_floating`(3), and `floating_to_decimal`(3).

The number of significant digits in implicitly-formatted FORTRAN list-directed output ( )`print*`,x,y,z has been increased so that sufficient decimal digits are printed to uniquely specify the binary value held internally.

`ieee_flags`

`ieee_flags`(3M) or `f77_ieee_environment`(3F) provide a uniform way of accessing the IEEE modes for rounding direction and rounding precision, and the IEEE exception-occurred accrued status bits, for Sun-3 with `-f68881` or `-ffpa`, and for Sun-4. Use `ieee_flags()` instead of `fpstatus_()` and `fpmode_()`.

`ieee_handler`

`ieee_handler`(3M) or `f77_ieee_environment`(3F) provide a uniform way of enabling IEEE traps on floating-point exceptions. Trapped exceptions result in SIGFPE signals. To conveniently exploit IEEE trapping, use `ieee_handler()` rather than `fpmode()`, `signal`(3), or `sigvec`(2).

Special IEEE Functions

`libm` now contains implementations of all functions needed for the IEEE Standard, its Appendix, or the IEEE Test Vectors, replacing the special functions listed in Appendix D of the SunOS 3.2 version of the *Floating-Point Programmer's Guide* manual. Use these (3M) functions or their `d_` or `r_` counterparts:

	IEEE Standard:
sqrt(3M)	square root
remainder(3M)	remainder
irint(3M)	convert floating-point value to integral value in integer format
rint(3M)	convert floating-point value to integral value in floating-point format
decimal_to_floating(3)	convert decimal record to binary floating point
floating_to_decimal(3)	convert binary floating point to decimal record
ieee_flags(3M)	get/set modes/status
ieee_handler(3M)	enable/disable trapping
	IEEE Appendix:
copysign(3M)	copysign
scalb(3M)	scalb - scale by base b to power n in floating-point form
scalbn(3M)	scale by base b to power n in integer form
logb(3M)	logb - exponent in floating-point form
ilogb(3M)	exponent in integer form
nextafter(3M)	nextafter
finite(3M)	finite
isnan(3M)	isnan
isnan(x)||isnan(y)	unordered
fp_class(3M)	class
	IEEE Test Vectors:
logb(3M)	L test vector
scalb(3M)	S test vector
significand(3M)	F test vector

Note that the <math.h> function `class()` and the <floatingpoint.h> struct member decimal_record.class , defined in the Sys4-3.2 release and some preliminary versions of SunOS 4.0, present an irreconcilable conflict with the "class" reserved word in C++ and other languages. Accordingly `class()`, `d_class_()`, `r_class_()`, and `decimal_record.class` are named `fp_class()`, `d_fp_class_()`, `r_fp_class_()`, and `decimal_record.fpclass` in the released version of SunOS 4.0.

Suppressing `ieee_retrospective` ***Warnings***

`ieee_retrospective` is a `libm` function invoked whenever a FORTRAN program terminates normally or abnormally for any reason (not necessarily because of a floating-point exception). Under IEEE arithmetic, exception-occurred bits accrue until explicitly cleared by the programmer. If any IEEE exceptions, other than inexact, occur but are not cleared by the time the program terminates, a message listing all the exceptions still uncleared is appended to standard error:

```
Warning: the following IEEE floating-point arithmetic exceptions
occurred in this program and were never cleared:
Inexact;  Underflow;
```

No message is printed if the only outstanding exception is inexact. Note that IEEE exceptions arise only on Sun-3 compiled with `-f68881` or `-ffpa`, or on Sun-4.

The significance of the warning may be difficult to determine. "Inexact" implies a normal rounding error, which is to be expected in floating-point programs, while "Underflow" and "Overflow" warn of rounding errors that may have been relatively larger than usual. Whether the underflowed or overflowed result affected the final answer can only be determined by analyzing the program. For instance, a small underflowed result may suffer significant relative rounding error, which matters if it is ultimately multiplied by a large number, but not if it is ultimately added to a large number.

The best way to suppress the message is to determine which operations are generating the exceptions, then altering the algorithm to avoid the exceptions. `ieee_handler`(3F) may be helpful in this search.

A second method is to clear all the exceptions at the end of the program by calling `ieee_flags`(3F). The `ieee_retrospective` message won't appear except possibly as part of an unplanned abnormal termination.

Finally the message can be fully suppressed by defining a function in the FORTRAN source

```
subroutine ieee_retrospective()
end
```

C programs do not call `ieee_retrospective` automatically. To obtain its effect, insert a call manually in a C program:

```
extern void ieee_retrospective_();

ieee_retrospective_();
```

While `abrupt_underflow` is enabled, exception handling need not conform to IEEE requirements. Thus in a program that enables `abrupt_underflow`, no message from `ieee_retrospective` does not imply that the results were not potentially corrupted by underflow. That possibility is presumed to have been eliminated by analysis prior to enabling `abrupt_underflow`.

Benchmarks

The following benchmarks indicate maximum performance for a Sun-3/280 compiled `-f68881` or `-ffpa`, and for a Sun-4/280. FORTRAN programs were compiled with `-O3`; the C program `spice3b1 -O2`; the Sun-supplied `libm.il` inline expansion templates were used. Benchmark sources are usually slightly modified for testing purposes; therefore comparisons to results obtained by others on non-Sun systems may be misleading.

The 100x100 Linpack benchmark results in KFLOPS pertain to FORTRAN code with ROLLED BLA's.

The 1000x1000 Linpack benchmark results in KFLOPS pertain to the best performance obtained from Fortran codes that do "higher-level granularity" unrolling of 4, 8, or 16 times.

The 100x100 Zlinpack benchmark results in complex-KFLOPS measure performance on a complex or doublecomplex arithmetic translation of the Linpack benchmark. A complex-KFLOP is equivalent to 4 real KFLOPS.

The double-precision Doduc benchmark results in elapsed SECONDS of real time measure performance on the non-linear nuclear reactor simulation benchmark distributed and reported by N. Doduc (`uunet.uu.net!mcvax!inria!ftc!ndoduc`).

The double precision FORTRAN and C versions of `SPICE`, `2G6`, and `3B1` respectively, measure elapsed SECONDS of real time for three representative decks: `EDGEREG`, a Schottky edge-triggered register, `COMPARATOR`, a differential comparator, and `DIGSR`, a CMOS digital shift register.

BENCHMARK RESULTS KFLOPS	3/280 -f68881	3/280 -ffpa	4/280
100 Linpack double	115	470	1070
100 Linpack single	125	900	1600
1000 Linpack double	160	600	1140
1000 Linpack single	180	1050	1700
100 Zlinpack complex	28	110	150
100 Zlinpack doublecomplex	26	80	170

BENCHMARK RESULTS ELAPSED SECONDS	3/280 -f68881	3/280 -ffpa	4/280
Doduc double	2000	880	530
spice2g6 EDGEREG	220	120	61
spice2g6 COMPARATOR	260	130	68
spice2g6 DIGSR	730	380	220
spice3b1 EDGEREG	210	100	61
spice3b1 COMPARATOR	260	125	72
spice3b1 DIGSR	810	390	270

The Sun-4 complex Zlinpack results are affected by the choice of inline expansion templates for complex multiplication. Double-precision products are used to maximize robustness against rounding errors and intermediate overflows and underflows, just as on the Sun-3 with

```
/usr/lib/f{68881,fpa}/libm.il
```

When such robustness is not needed, performance can be significantly improved by substituting a user-coded single-precision complex multiplication template.

The Linpack benchmarks incorporate a slight modification in the distributed `EPSLON` routine. Without the modification the normalized residual is not computed correctly when compiled with `-f68881 -O3` without `-fstore`. The modifications to `EPSLON` don't affect the benchmark itself or the computation of the actual residual, only the scaling factor applied to compute the normalized residual. The modifications are necessary as a result of improvements in the global optimizer, which is now able to allocate more variables to registers. The following extract reveals the modification in lower case:

```
      REAL FUNCTION EPSLON (X)
      REAL X
      REAL A,B,C,EPS
C
C     THIS PROGRAM SHOULD FUNCTION PROPERLY ON ALL SYSTEMS
C     SATISFYING THE FOLLOWING TWO ASSUMPTIONS,
C        1.  THE BASE USED IN REPRESENTING FLOATING POINT
C            NUMBERS IS NOT A POWER OF THREE.
C        2.  THE QUANTITY  A  IN STATEMENT 10 IS REPRESENTED TO
C            THE ACCURACY USED IN FLOATING POINT VARIABLES
C            THAT ARE STORED IN MEMORY.
C     THE STATEMENT NUMBER 10 AND THE GO TO 10 ARE INTENDED TO
C     FORCE OPTIMIZING COMPILERS TO GENERATE CODE SATISFYING
C     ASSUMPTION 2.
C
      A = TOREAL(4)/TOREAL(3)
        call dummy(a)
   10 B = A - ONE
      C = B + B + B
      EPS = ABS(C-ONE)
      IF (EPS .EQ. ZERO) GO TO 10
      EPSLON = EPS*ABS(X)
      RETURN
      END

        subroutine dummy(a)
        REAL a
        end
```

The `-fstore` option may adversely affect performance and is usually useful only in code very much like `EPSLON` which attempts to determine properties of machine arithmetic.

MC68881 Mask Differences

New MC6888X varieties

`mc68881version`(8) has been expanded to differentiate B96M MC68882's and B81G MC68881's as well as the A79J and A93N MC68881's previously distinguished. Future Sun products may incorporate B81G MC68881's and B96M 68882's, so it is worthwhile reviewing the differences. Note that chips which `mc68881version` classifies as "A93N" or "B81G" may be physically marked as a different, functionally equivalent, mask set.

- A79J MC68881's are the original chips shipped in early Sun-3's. They have not been shipped by Sun since mid-1986. Their bugs are listed in an appendix to the SunOS 3.2 version of the *Floating-Point Programmer's Guide* manual.
- A93N MC68881's are the standard chips in most Sun-3's. They have one known bug: extended precision addition and subtraction occasionally round the wrong way when the correct result is very nearly half-way between two representable extended-precision numbers. This bug is almost undetectable on Sun systems since extended precision data types are not directly available in compiled languages. The bug may be observed by running IEEE test vectors directly on extended-precision operands and results through assembly-language coding; it also shows up in the computed residual of the Linpack benchmark program and may affect any other computed result which is essentially roundoff noise. Since most application programs avoid printing out such results, they aren't much affected by the bug.
- B81G MC68881's are identical to A93N's except that the bug has been fixed. As of the end of 1987, no Sun products used B81G's.
- B96M MC68882's are functionally identical to B81G MC68881's in user mode. Because they implement some parallel execution, their internal state is more complex than MC68881's, so the size of the information dumped by the privileged `FSAVE` instruction is greater. Consequently MC68882's can't be used on Sun operating systems prior to 4.0. As of the end of 1987, no Sun products used MC68882's.

Suppressing A79J Warnings

In order to improve performance of the correctly-functioning later masks, SunOS 4.0 no longer attempts to work around the shortcomings of the early A79J MC68881's. The 3.2 manual describes some of these shortcomings, and mentions that some have no software workaround anyway.

SunOS 4.0 will print a warning message on standard error if a program compiled `-f68881` is executed on a Sun-3 with an A79J MC68881:

NOTE *Please note: MC68881 upgrade to A93N mask may be advisable. See Floating-Point Programmer's Guide.*

A program that does not require extensive floating point can bypass the message by recompiling with `-fsoft`.

If it can be determined that none of the A79J shortcomings affects a particular program, then the message can be suppressed by including a C function

```
int ma93n_()
{
return 1;
}
```

or a FORTRAN function

```
integer function ma93n()
ma93n=1
end
```

or by modifying the executable image with `adb`(1).

System V Interface Compliance

Unlike SunOS 3.2, SVID compliance on SunOS 4.0 is the same for all Sun-3 floating-point code generation options and for Sun-4. Compliance is not claimed for certain aspects of SVID that are contrary to the intent of the IEEE Standard. See `matherr`(3M).

The `libm.il` inline expansion templates do not call `matherr`(3M) or set `errno` and therefore should not be used if detailed SVID compliance is required.