Implementation of IEEE Floating-point Arithmetic on the Cray T90 System

James M. Kiernan, Sun Microsystems, Inc. and
William J. Harrod, Cray Research, Inc.

ABSTRACT: *Cray Research will offer a version of the CPU for the Cray T90 system that implements a subset of the IEEE Standard 754 for floating-point arithmetic. A description of the 64-bit IEEE arithmetic implementation on the Cray T90 system with IEEE floating point hardware architecture is presented. Instruction set differences between this system and machnes with traditional Cray arithmetic are discussed, as well as the trade-offs necessary to maintain high performance on a vector supercomputer. Some examples of programming style will be given that take advantage of vectorized IEEE arithmetic.*

1 Introduction

The Cray T90 system with IEEE floating point hardware will be the first parallel-vector supercomputer (PVP) offered by Cray Research to support the IEEE Standard 754 for floating-point arithmetic[1]. The machine's architecture will conform to the Cray Research IEEE definition as described in a previous CUG paper [2]. While the initial version of the Cray T90 system implements the traditional Cray floating-point arithmetic, the changes required for the Cray T90 system with IEEE floating point hardwares T90 system with IEEE floating point hardware were designed to be confined entirely to the CPU so that existing Cray T90 systems could be upgraded to a IEEE system by swapping CPU boards. No changes to memory, SSD, disk or I/O subsystem are necessary. It is important to note that the Cray T90 system with IEEE floating point hardware processor is a redesigned CPU, with many different op-codes and new instructions, and is not binary compatible with any previous CRI machines. In particular there is no Cray C90 compatibility mode, nor does the Cray T90 system with IEEE floating point hardware have dual mode (i.e. Cray & IEEE) floating-point functional units. Therefore, existing Cray C90 or T90 system binaries will not execute on a Cray T90 system with IEEE floating point hardware due to arithmetic and instruction set differences. As will be seen below, there was more involved in implementing IEEE arithmetic than merely replacing the floating-point add/multiply functional units.

It should be noted that the hardware implements 64-bit (IEEE double) floating-point arithmetic only. Cray T90 system with IEEE floating point hardware 128-bit floating-point arithmetic will continue to be done via a software implementation compatible with the Sun quad format, although this does not rule out the possibility of full 128-bit arithmetic support in future hardware. The new 128-bit format is also not compatible with the previous CRI 128-bit arithmetic, although 64-bit IEEE format is compatible between the Cray T90 system with IEEE floating point hardware and Cray T3D/E systems.

In this technical paper we will use the term "Cray T90 system" to denote a Cray T90 system with IEEE floating point hardware.

2 Basic Rationale

The goal of the Cray T90 system implementation is to maximize the support of IEEE arithmetic with negligible impact on performance. In practice, this means that currently existing Cray application software should port to the Cray T90 system with minimal effort, and that existing applications which run on other IEEE machines such as workstations should also port to the Cray T90 system easily. Minimal effort means that a numerically robust code on an IEEE machine should be equally well behaved on the Cray T90 system, and that in many cases recompiling the source will be sufficient. Assembly language codes which use floating-point instructions will have to be checked, and possibly modified, along with older CRI codes that perform Boolean tricks with, or machine dependent manipulations of, floating-point numbers.

To achieve minimal performance penalty in conversion to IEEE arithmetic, one of the goals for the Cray T90 system was to ensure that the fundamental floating-point operations of addition and multiplication run in both scalar and vector mode, with a latency within 1 CP of traditional CRI arithmetic. To maintain reasonable performance it was clear that true division (rather than reciprocal approximation) would need to be done via hardware, along with the floating-point comparison operation. It was also desirable to have hardware support for conversion between floating-point and integers to avoid convoluted software to handle the IEEE special cases such as overflows, NaNs, and infinity.

Because vector machines do not have precise, recoverable traps it was not possible to simply ignore the many IEEE special cases and fix-up the exceptions after the fact via software, which is a common method for workstations. That would have slowed scalar algorithms with trap-handling code, and would still have been impossible in vector pipelines, which are uninterruptible. Instead a clean implementation of the IEEE- recommended "sticky" bits, which record the occurrence of exceptional conditions, was added to the hardware, along with bits to set the current rounding mode. There are also user-settable bits which enable or disable interrupts on each possible exception. All these floating-point status bits are stored in a register which is also part of the exchange package. The rounding mode bits are read dynamically as each floating-point instruction issues, and are not embedded in the instruction op-codes.

3 Summary of CRI IEEE Definition

The CRI IEEE definition includes most of the required features in the IEEE Standard 754 . The implementation includes:

- 64-bit binary format for floating-point data (called double in the IEEE standard; Cray Fortran type REAL; C type double)
- signed zero (± 0)
- infinity (±∞)
- quiet and signaling NaNs
- the basic arithmetic operations of add/subtract, multiply, divide, remainder and square root,
- four rounding modes
- 5 IEEE exceptions: overflow, underflow, div-by-zero, invalid, inexact result
- IEEE comparisons, including comparison involving zeroes, NaNs and infinity.

There are two areas of nonconformance to the IEEE Standard 754 , namely the lack of support for the 32-bit IEEE data type and for "denormal" numbers. The absence of these two features on the Cray T90 system does not preclude support on future systems.

Software support is provided for 32-bit (IEEE single) data for reading and writing, and conversion to and from 64-bit (IEEE double) format, via the I/O library. This allows 32-bit data to be shared with workstations, but all Cray T90 system arithmetic operations (on 32-bit imported data) will be 64-bit only. 128-bit data (Cray Fortran type DOUBLE PRECISION;C type long double) is supported via a software library only, and uses the same format as Sun quad precision. Denormal or subnormal arithmetic (sometimes called gradual underflow) is not supported for performance and architectural reasons, since the lack of precise traps prevents handling denormals via trapping software, and the limitations of space in the arithmetic functional units prevents full denormal support at present. Floating-point functional units treat denormals as zero on input and flush would-be denormals to zero on output.

4 Review of the Cray IEEE Implementation

4.1 *Basic Arithmetic*

The basic arithmetic operations for floating-point include add, subtract, multiply, and divide, with instructions for the usual combinations of scalar<op>scalar, scalar<op>vector, and vector<op>vector operations (Table 1)

Table 1. Elementary Floating-Point Instructions

Operation	*Instruction*	
Addition	Si	Sj+FSk
	Vi	Sj+FVk
	Vi	Vj+FVk
Subtraction	Si	Sj-FSk
	Vi	Sj-FVk
	Vi	Vj-FVk
Multiplication	Si	Sj*FSk
	Vi	Sj*FVk
	Vi	Vj*FVk
Division	Si	Sk/FSj
	Vi	Vk/FSj
	Vi	Vk/FVj

Note that the "/F" instruction is a true divide operation, and the older CRI reciprocal (Newton iteration) instruction sequence is no longer supported. Also the older multiply variations such as half precision multiply (i.e the "*H") and truncated vs. rounded multiply (i.e. "*R") are gone, along with the scalar instructions which performed load-immediates of certain special values of floating point constants (Si .6; Si 1.; Si .4 etc.).

4.2 *Additional Operations*

Some other frequently used operations (Table 2) which also have new hardware instructions include square root, real to integer conversion, real to rounded integer, and integer to real. The INT instruction truncates a real to an integer as specified by the Fortran INT function, whereas the RINT instruction will round to an integer according to the current IEEE rounding mode in effect. The real to integer conversions support full 64-bit signed integers (2's complement). The SQRT instruction uses the divide functional unit, and returns the result rounded as specified by IEEE Standard 754.

Table 2. Extended floating-point instructions.

Operation	Instruction	
square root	Si	SQRT,Sj
	Vi	SQRT,Vj
float to int	Si	INT,Sj
	Vi	INT,Vj
float to rounded int	Si	RINT,Sj
	Vi	RINT,Vj
int to float	Si	FLT,Sj
	Vi	FLT,Vj

4.3 Comparison

Because of requirements of the IEEE standard specifications, it was necessary to have floating-point compare (Table 3) done via a hardware instruction. All the comparisons used in standard Fortran, plus an unordered compare, are implemented directly in hardware.

Table 3. Floating Point Compare Instructions

Operation	*Instruction*	
Equal	Si	Sj,EQ,Sk
	VM	Sj,EQ,Vk
	VM	Vj,EQ,Vk
Not equal	Si	Sj,NE,Sk
	VM	Sj,NE,Vk
	VM	Vj,NE,Vk
Greater than	Si	Sj,GT,Sk
	VM	Sj,GT,Vk
	VM	Vj,GT,Vk
Greater than or equal	Si	Sj,GE,Sk
	VM	Sj,GE,Vk
	VM	Vj,GE,Vk
Less than	Si	Sj,LT,Sk
	VM	Sj,LT,Vk
	VM	Vj,LT,Vk
Less than or equal	Si	Sj,LE,Sk
	VM	Sj,LE,Vk
	VM	Vj,LE,Vk
Unordered	Si	Sj,UN,Sk
	VM	Sj,UN,Vk
	VM	Vj,UN,Vk

By implementing the comparison in this manner, it was possible to avoid having to change or create any additional jump instructions or jump logic (such as “jump on floating unequal”, etc.) This keeps the current jump and instruction buffer logic unchanged.

For scalar compare, such as

```
S0      Sj,EQ,Sk
JSZ     LABEL1      ;go here if compare is false
JSN     LABEL2      ;go here if compare is true
```

if the comparison is TRUE, then the result register Si is filled with 1 bits, otherwise Si is all 0 bits. The Si register can then be used as a bit mask, or in a jump on zero/non-zero, or a jump on plus/minus (i.e the sign bit). For vector comparisons, such as

```
VM      Vj,EQ,Vk
```

a bit within VM is set to 1 if the compare involving the corresponding pair of elements of Vj and Vk is TRUE, otherwise the VM bit is set to 0.

The *unordered* comparison is TRUE (1) if either operand (or both) is a NaN, otherwise it returns FALSE (0). Under the rules of IEEE, a NaN never equals anything, including itself. The unordered compare operation is not defined in standard Fortran or C language syntax. In CAL, to test whether a number X is a NaN, it may be “compared to itself” via the EQ or UN comparisons. This can be used to implement a fast (and vectorizable) isnan(x) function, which returns TRUE if the argument x is a NaN, and FALSE otherwise. While X could also be compared to itself using Fortran or C, an optimizing compiler would probably restructure such statement blocks as:

```
IF(X .EQ. X) THEN
   (code for normal X here)
ELSE
  (code if X=NaN)
ENDIF
```

and just generate code for the case where X=X. This style of coding is not recommended due to portability problems among different vendors and different compiling systems, some of which may allow expressions like X .EQ. X, while others optimize it away.

The recommended implementation is as follows:

```
IF(isnan(X)) THEN
   (code if X=NaN)
ELSE
  (code for normal X here)
ENDIF
```

The following is a coding example (in pseudo-CAL) of how a Fortran MAX function might be implemented (without jumps) using the new compare instructions.

```
(s1 == max(y,z))

s2      y           ;value for y
s3      z           ;value for z
s1      s3
s7      s2,gt,s3    ;all 1's if y > z
s1      s2!s1&s7    ;max (scalar merge)
```

A vectorized version of the same function is similar:

```
(v1 == max(y,z))
v2      y           ;value for y
v3      z           ;value for z
```

```
vm      v2,gt,v3      ;VM=1 where y(i) > z(i)
v1      v2!v3&vm      ;max (vector merge)
```

4.4 Sqrt/Divide Functional Units

The divide functional unit is internally segmented into multiple div/sqrt subunits, each of which computes a quotient or square root from its input operand(s). For vector and scalar pipelined operations, the next pair of operands are sent to the next available internal subunit, so that, when viewed from the "outside", the entire divide/sqrt operation appears to be pipelined (Figure 1).

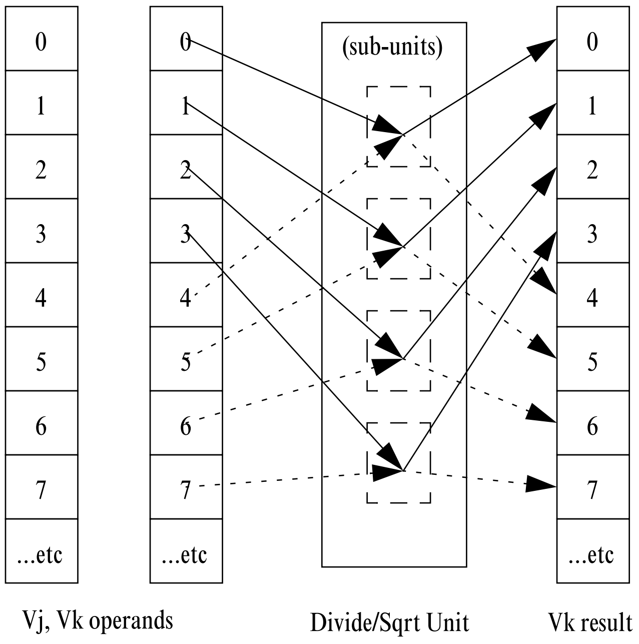

Figure 1: Divide/Sqrt Functional Unit

The figure is somewhat simplified as the actual number of subunits may not be 4; also, the Cray T90 system is a dual pipeline machine so all the functional units are doubled in odd/even pairs, with the 128 elements of each vector register being read/written in odd/even pairs to dual functional units. The result for the Cray T90 system is that a vector add, subtract or multiply will deliver 2 vector results every CP (plus start-up latency), while the divide or square root will deliver 1 result every 2 CP's (plus start-up latency).

Other, non-elementary floating-point operations which are not implemented directly in hardware include binary to decimal conversion and 32-bit to 64-bit real conversion. These are both handled through library support. The IEEE "remainder" function can be implemented either via compiler generated in-line code or by a library function.

5 Floating-point Status Bits

The Cray T90 system implementation also includes support for detecting IEEE exceptional conditions (such as X/0, $\sqrt{-4}$, 3.0 + NaN, ∞/∞, etc.) and for user control of interrupts on floating-point exceptions. Some additional support, beyond that required by IEEE, is also available for assistance in running older applications. There are 14 bits (Table 4) which define the user's "floating-point state". These bits are available via instructions that read and write Status Register 0 (SR0) and are automatically saved and restored for each CPU via the hardware exchange package when jobs or tasks are switched.

Table 4. Floating Point Status Bits

Type	Purpose	value
Exception	Overflow	1_2 or 0_2
	Underflow	1_2 or 0_2
	Divide by 0	1_2 or 0_2
	Invalid	1_2 or 0_2
	Inexact	1_2 or 0_2
(CRI)	Exc. Input	1_2 or 0_2
Interrupt	Overflow	1_2 or 0_2
	Underflow	1_2 or 0_2
	Divide by 0	1_2 or 0_2
	Invalid	1_2 or 0_2
	Inexact	1_2 or 0_2
(CRI)	Exc. Input	1_2 or 0_2
Rounding	To nearest	00_2
	To +∞	01_2
	To 0	10_2
	To −∞	11_2

5.1 Exceptions: the "sticky bits"

The first set of 6 bits (the "sticky" bits) are initialized to zero at the beginning of the user's program, and record the subsequent occurrence of exceptional operations. As specified by the IEEE standard, these bits can only be cleared by the user. A CRI extension to the standard provides an "exceptional input operand" bit, which will record the use of ∞ or NaN. This allows these numbers, which may have been preset to represent uninitialized data (by the loader, for example), or which have been generated quietly during program execution, to trigger an exception or interrupt only when they are actually used in a floating-point operation. A value of 0 for any of the "sticky" bits means that the corresponding exception has not occurred (since the bit was last cleared); a value of 1 means the exception has occurred.

5.2 Interrupt enable/disable

The second set of 6 bits (the interrupt enable/disable bits) are also initialized to zero at the beginning of the user's program if running in the IEEE-conforming mode. Each interrupt bit corresponds to one of the exceptional conditions recorded by the "sticky" bits above. A value of 0 means that the corresponding interrupt is disabled. A value of 1 means that if the corresponding exception occurs, an (imprecise) interrupt will be generated, and can be handled similarly to the way that current CRI C90 machines work (i.e. via the UNICOS signal mechanism). Note that when an interrupt is disabled, then the program will run without notifying the user when that exception has occurred, unless the user checks the status of the corresponding "sticky" bit.

The exception and interrupt bits should not be read or written by the user while the floating-point functional units are busy. Doing so may cause an exception to be "missed". A special instruction, Complete Floating Point (CFP), was added to the Cray T90 system to force an instruction hold-issue until all current floating-point operations are beyond the point at which exceptions would occur.

5.3 Rounding mode control

The final set of 2 bits determine the current IEEE rounding mode for all floating-point operations. The combination of 2 bits gives the 4 possible IEEE modes, with round-to-nearest (00_2) being the default. These bits are sensed at the start of each floating-point instruction, and "remembered" for the duration of that operation. Unlike the exception/interrupt bits above, changes to the rounding mode bits (SR0) can be made immediately after a floating-point operation has begun, and the new rounding mode takes effect at the next floating-point instruction. Currently executing operations continue unchanged.

5.4 User Control

A program running in the IEEE Default mode begins executing with all the above 14 bits cleared to zero. But the user can override these defaults, and also sense, clear and set the bits during program execution. Instructions are provided which read and write these bits (Table 5).

Table 5. Read/write Floating-Point Status.

Operation	*Instruction*	
read status register	Si	SR0
write status register	SR0	Si
enable invalid interrupts	EFI	INV
disable invalid interrupts	DFI	INV
enable div-by-0 interrupts	EFI	DIV
disable div-by-0 interrupts	DFI	DIV
enable overflow interrupts	EFI	OVF
disable overflow interrupts	DFI	OVF
enable underflow interrupts	EFI	UNF
disable underflow interrupts	DFI	UNF
enable inexact interrupts	EFI	INX
disable inexact interrupts	DFI	INX
enable except. input interrupts	EFI	INP
disable except. input interrupts	DFI	INP

Table 5. Read/write Floating-Point Status.

Operation	*Instruction*	
set round to nearest mode	RNM	
set round to zero mode	RZM	
set round up (to +∞ mode	RUM	
set round down (to -∞ mode	RDM	
set round mode from Si	SETRM	Si

The first two instructions read or write the entire status register SR0 using an S register, and take 1 CP. Any bit or group of bits can then be manipulated using Boolean mask operations on Si. These instructions can also be used to save the floating-point state upon entry to a subroutine, for example, and then restore the state on return. The next group of 12 instructions allow the interrupt enable bits to be set individually with one operation. The next 4 instructions set the rounding mode bits directly, without the need to read SR0 or wait for on-going floating-point operations to complete. These instructions take 1 CP to issue and can be used for fast switching of rounding modes, for example, in interval arithmetic. The final SETRM instruction allows the 2 rounding mode bits to be set directly from an S register, without the need to read SR0 and mask the rounding bits in.

Caution should be taken when manipulating SR0 directly since its 64 bits contains other information affecting the user job in addition to the 14 floating-point status bits. There will be a higher level library interface to the mode bits to improve code portability; however users should note that the precise mechanism or language binding for manipulation of IEEE exception, interrupt, and rounding modes was never a part of the IEEE Standard 754, and thus varies among different computer vendors.

6 Performance Issues

The Cray T90 system with IEEE floating point hardware was designed with supercomputer-level performance in mind. Since the basic IEEE arithmetic operations (+ – * / √) and comparison vectorize, most existing Cray applications should continue to vectorize without source code changes. Keeping the floating-point status bits in a register (i.e. dynamic control rather than within the op-codes) allows external routines such as LIBSCI, LAPACK, IMSL, and NAG libraries to be built once, and possibly called with different modes to investigate issues of numerical stability. For example, if a numerical instability is suspected in an application, a matrix manipulation subroutine could be called multiple times, but with different rounding modes in effect, and the results checked for unusually large numerical differences.

6.1 User control of floating-point state

All the IEEE mode bits are directly accessible by the user without any intervention (and overhead) by the operating system. By running with the interrupts for division by zero, overflow, and invalid enabled, existing Cray codes can continue to exhibit the expected behavior when errors such as division by zero occur. However, routines can also be written that execute with interrupts disabled (IEEE "default mode"), and exceptions can be discovered by periodic (and relatively cheap) inspection of the "sticky" bits, no precise, recoverable (and expensive, from a performance standpoint) traps are necessary. As compiler and library support are added for detecting some common IEEE special cases, it will also be possible to perform some limited manipulation of NaN and ∞ while still achieving high vector performance. Depending on the algorithm involved, it is possible for applications to run nonstop, while generating, sensing, and replacing NaNs and infinities with more benign values on-the-fly. This section includes some simple examples and suggestions for maintaining high performance while using some features of IEEE arithmetic.

6.2 *Functional Units and Chaining*

One of the easiest ways to take advantage of the Cray T90 system vector capabilities is to maximize the use of chaining between multiple independent functional units. Table 6 shows which floating-point operations occur in each functional unit.

Table 6. Floating Point Functional Units

Unit	Operation
Add	X+Y
	X-Y
	FLT
	INT
	RINT
	COMPARE
Multiply	X*Y
	I*J (64x64-bit upper result)
	I*J (64x64-bit lower result)
Divide	X/Y
	$\sqrt{X}$

Note that the other vector units (Integer, Boolean, Pop Count, etc.) still exist and are unchanged on the Cray T90 system with IEEE floating point hardware. The divide functional unit can be exploited by the compiler for better code scheduling, since the unit uses at most 3 vector registers, and can run in parallel with, or chain into, the add and multiply units. This contrasts with the older reciprocal iteration sequence on Cray machines, in which most of the Newton iteration used the multiply unit and additional vector registers.

Another way to exploit the divide unit is to compute X/Y as X*(1/Y) when numerical accuracy allows it (Figure 2). In the rational polynomial evaluation, assume that replacing the true quotient with the reciprocal is acceptable. The code sequence is typical of elementary function routines such as sin(x). Each step of polynomial evaluation represents one vector "chime", i.e. a chain of pipelined vector operations. We begin by computing

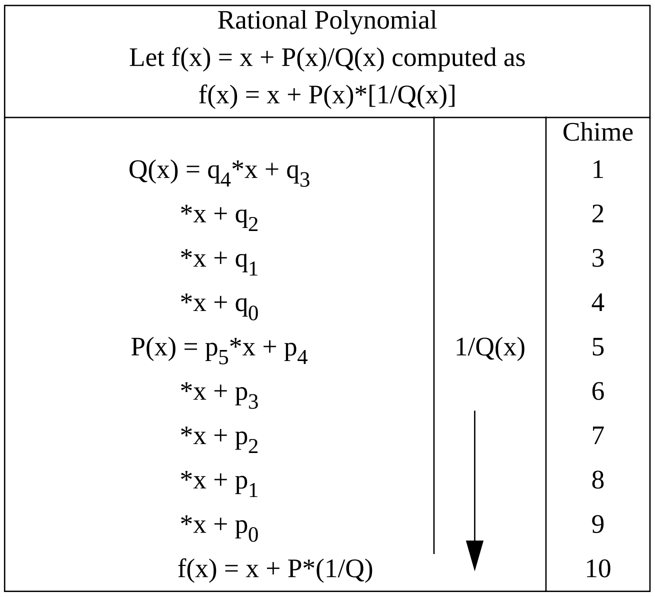

Figure 2: Overlapped divide, add/multiply

the "denominator" polynomial Q(x). In chime 5, the reciprocal is begun in the divide unit, while the add and multiply units are still free to continue with the "numerator" polynomial P(x). The result 1/Q(x) is ready by the time P(x) is finished, and chains into the final result f(x) in chime 10. Note that while the divide may have a slower start-up latency time than either add or multiply, and takes 4 chimes to complete, instruction scheduling in this case allow the divide (reciprocal) to be "free". This code sequence would consume several more chimes if we computed P(x), then Q(x), and then P/Q. It also would be slower if we used the older Cray reciprocal iteration sequence, which needs the multiply unit to complete the Newton iteration, adding several more chimes to the process.

6.3 *Special cases: Infinity and NaN*

One of the problems associated with IEEE arithmetic, from a performance aspect, is dealing with the special cases or exceptions such as infinity and NaNs. There is no universal method of handling these, as the actions to be taken depend on the algorithm. However on the Cray T90 system there are several ways of dealing with these end-cases that still allow vectorization to occur. There are at least 3 techniques which can be used to deal with conditional If-Then-Else constructs within loops, depending on the complexity of the algorithm involved.

6.3.1 *Simple If-Then-Else Blocks*

Consider the case for simple conditions within loops such as:

```
loop
  if (condition) then
    (block-1)
  else
    (block-2)
  endif
endloop
```

where block-1 and block-2 are both likely to be executed, and both are equally expensive in terms of time. Because vector operations are relatively cheap the first method for dealing with this (whether the condition is related to a special case of IEEE arithmetic or not) is just to store the result of the conditional test in the VM register, execute both blocks for all the data, and then do a vector merge at the "endif" statement to select the correct results from each block. This also assumes that the condition expression is vectorizable.

Here are some examples using this approach to do vectorizable replacements of ∞ and NaNs in the above type of conditional code blocks. There is compiler/library support for the IEEE recommended functions like isnan(x) and isinf(x). In Fortran these are LOGICAL functions, returning true if their respective arguments are NaN or ±∞, and otherwise false. By generating code that uses the vector compare instructions, in-line expansion of these functions vectorize.

Example 1

In the following example, suppose that an algorithm can generate ∞ on some occasions, but that it could be replaced by some other, very large number (such as 10^{100}) without affecting the numerical results of the application. Since the range of IEEE double values ($100^{\pm 308}$) is more restricted than the older Cray arithmetic ($10^{\pm 2465}$) this situation is not unrealistic for some existing Cray applications. The Fortran vector loop below could be used to filter out the spurious ∞'s and replace them with a more well-behaved value.

```
do i=1,n
  temp = y(i)**4    !this may overflow
  if(isinf(temp) then
    x(i) = 10^100
  else
    x(i) = temp
  endif
enddo
```

The reason that forcing a value of 10^{100} is safer than keeping the ∞ is that if the x(i) is subsequently involved in any subtractions, $10^{100} - 10^{100} = 0$, whereas $\infty - \infty =$ NaN under IEEE rules. Any generated NaNs can spread throughout the program like the plague. Since the above case only involved ∞, and we know that compares involving ∞ work correctly in the hardware, we could also have used the Fortran MIN function.

```
do i=1, n
  x(i) = y(i)**4    !this may overflow
  x(i) = min(x(i) , 10^100)
continue
```

Example 2

Another example involves a user-written atan2(y,x) function. Suppose that for performance reasons a user needs a less precise (but faster) atan2(y,x) function than the one supplied with the math library. The following pseudo-Fortran code fragment will handle the special cases of atan2(y,0), and atan2(0,0) (and some others) on the fly. Assume that atan2(0,0) = 0 is acceptable, and use the Cray-specific vector-merge intrinsics to replace y/x in cases such as y/0, y/x = overflow, 0/0, x = NaN, y = NaN, etc. The use of such functions reduces program portability, but may generate more efficient loop code, especially if we can avoid nested if-then-else statements, which are more difficult to vectorize.

```
  real x(n),y(n),z(n)
  logical inf,nan
c
c compute z = atan2(y,x)
c by some (vectorizable) user method
c
  do i = 1,n
    arg = y(i)/x(i)
    inf = isinf(arg)
    nan = isnan(arg)
    safearg = cvmgt(1.0,arg, inf.or.nan)
    answer = atan_user(safearg)
c
c handle case along y-axis
c
    answer = cvmgt( π/2 ,answer,inf)
c
c handle case at origin
c
    z(i) = cvmgt(0.0,answer,nan)
  continue
```

Note the cvmgt(arg1,arg2,arg3) function returns its first argument if the third argument is true, otherwise it returns the second argument. The final two uses of cvmgt could be merged into one, and overlapped with the computation in the user function atan_user, thus saving a chime, but that would tend to clutter up the example.

6.3.2 More complex if-then-else structures

Now consider the case for more complicated situations involving conditional blocks within loops

```
loop
  if(condition) then
    (block-1)
  else
    (block-2)
  endif
endloop
```

where perhaps both blocks are very expensive to compute, or where the condition tests for a rare occurrence such that block-1 almost always is executed, but block-2 practically never. We could use the gather/scatter method, but that requires several trips to memory in order to separate the elements required for block-1 and block-2. Also it might be that in this case simple substitution of normal numbers in place of NaNs or ∞ is not sufficient. Perhaps block-2 is slow, expensive, or does not vectorize (that situation might require the use of CAL.

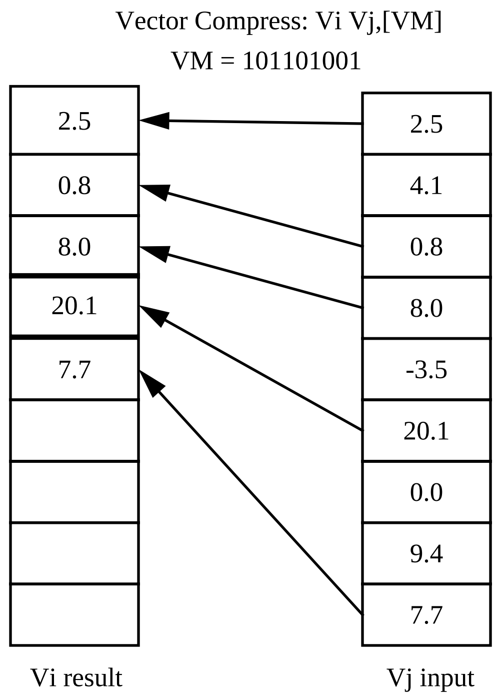

Figure 3: Cray T90 system vector compress using vector mask.

Example 3

The Cray T90 system with IEEE floating point vector compress and expand instructions (Figs. 3 & 4) may be used to advantage, and to keep the performance high for the most common (i.e. non-exceptional) cases. The vector-to-vector compress/expand instructions are new to the Cray T90 system with IEEE floating point, but are not limited to uses involving IEEE arithmetic.

The vector compress (Figure 3) instruction selects those elements of the input vector (Vj) whose corresponding bit in VM is set and places them into the result vector (Vi).

The vector expand instruction (Figure 4) uses successive elements of the input vector, and places them in the position in the result vector (Vi) whose corresponding bit in VM is set. Elements in Vi whose VM bit is 0 are not modified (the XXXX elements in Figure 4). The count for both compress and expand is determined by the number of bits set in VM, not by the vector length in VL.

Suppose a Fortran structure looks like

```
    loop
      if(x(i) .lt. maxarg .and.
*        x(i) .gt. minarg) then
        (normal case)
        y = user_func(x(i))
      else
        (very rare case)
        (could be complicated)
```

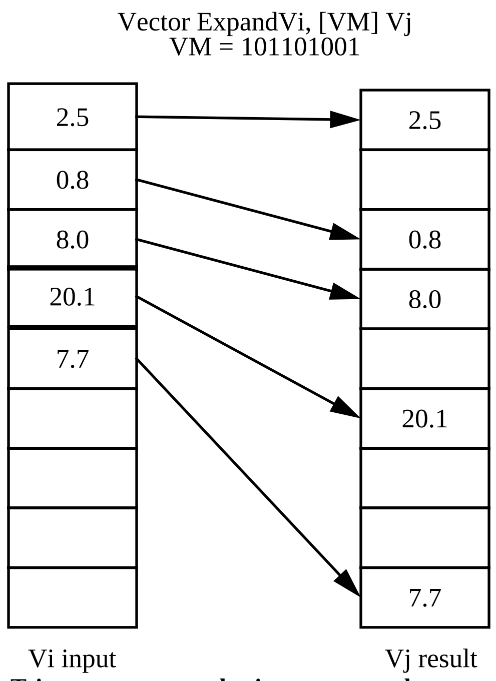

Figure 4: Triton vector expand using vector mask.

```
        endif
      endloop
```

Note the construction of the comparison tests such that the normal x(i) fall though, but x(i) which are outside the desired range of (minarg,maxarg), including ∞ and NaN, will not. Also, the normal (most common) case is coded first, since falling through a jump is faster than taking it. The following pseudo-CAL version of this example could vectorize for the normal case

```
  If (normal-condition)
    Y(i) ← X(i)compress normal X
    shorten VL, if needed use popcnt (VM)
    compute W(i) ← func(Y(i))
    Z(i) ← W(i)!expand
    restore original VL
  Else
    complement VM
    Y′(i) ← X(i) !compress abnormal X
    shorten VL
    compute W′(i) ← func(Y′(i))
    Z′(i) ← W′(i)!expand
    restore original VL
    restore original VM
    (vector merge results together)
    Z(i) ← Z(i)!Z′(i) & VM
  Endif
```

6.4 Difficult special cases

The last method of handling IEEE exceptions at reasonable performance occurs in cases where the occurrence of overflows, etc. either cannot be cheaply anticipated and tested for in advance, or where the handling of these conditions cannot be done within the confines of the DO LOOP structure. Such cases could require re-scaling the data (to prevent an overflow), or printing error messages and prompting an interactive user for assistance, or branching back to a previous program checkpoint and restarting a portion of the application using a different or more robust algorithm. One possible approach to these situations is given in Figure 5.

Note we are still assuming that exceptions are “exceptional”, that is, relatively rare, and we wish to avoid penalizing the algorithm with overly cautious amounts of testing for conditions that almost never happen. The approach here is to divide the most intensive portion of the code into moderately sized pieces, and execute a “quick & dirty” algorithm that presumably will succeed most of the time. Then, periodically, outside of the innermost loop (so it will still vectorize), test the IEEE exception bits to see if any overflows, div-by-zero, invalid operations etc. have occurred. If they have, then the program can branch off somewhere and attempt to handle them, and possibly resume the computation at some point. In the worst case, the user could at least exit the program gracefully. If checking the IEEE status bits indicates no exceptions have occurred, then the next pass through the inner, vector loop can proceed. Since reading SR0 is fast, comparable in overhead to reading the real-time clock, the latter case represents the “fast path” through the code.

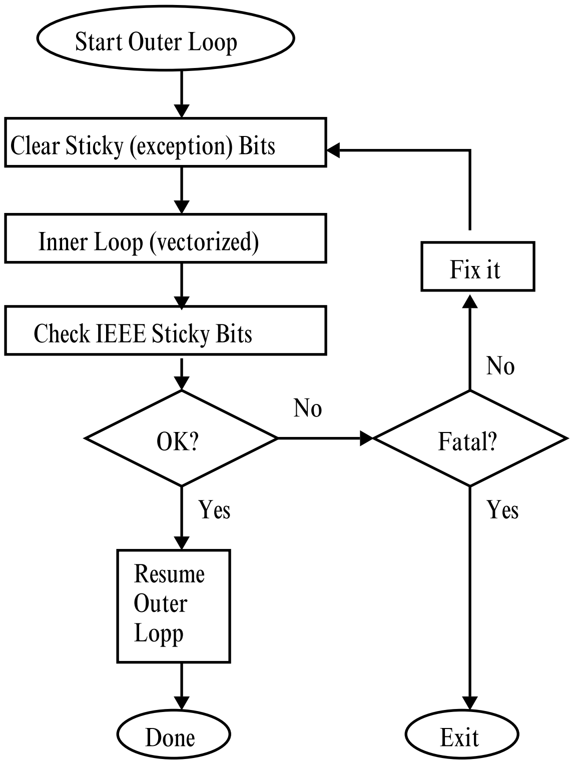

Figure 5: Special Cases

Because the Cray architecture can perform at high computational rates even for relatively short vector lengths, a loop of say, 10^4 elements could be broken into 10 groups of 1000, and still run at almost the original speed, provided that no exceptions occur, which is to say, most of the time. The exact method of restructuring the code is beyond the scope of this paper as it involves trade-offs between the size and complexity of the innermost loop, the expected frequency of exceptional occurrences, and the expense of “backtracking” in the code to determine the reason for the exception(s) that occurred.

A specific use of the method of Figure 5 that occurs in practice involves the standard library function A = CABS(Z), the complex absolute value, where Z = X + iY. The fastest method to evaluate this,

$$a = \sqrt{X^2 + Y^2}$$

will work safely most of the time, but gets into trouble whenever the intermediate quantities X^2, Y^2 overflow or underflow. While these cases are rare, the math library must anticipate them, and add some safe scaling for all arguments, thus slowing down the algorithm for everyone. By taking advantage of the IEEE sticky bits, we can speed up the results for the normal cases, and only perform the extra scaling when exceptions actually occur. Thus we can code as

```
(clear sticky bit for overflow)
do i=1,n
  a_i = sqrt(x_i**2 + y_i**2)
enddo
{wait for floating-point quiet (CFP)}
if (overflow bit set) then
  do i=1,n
    a = max (x_i, y_i)
    b = min (x_i, y_i)
    a_i= a*sqrt(1. + (b/a)**2)
  enddo
endif
```

While the example is in pseudo-Fortran, in an actual CAL math library routine the “loop length” is always ≤ 128, so little extra work is required if the overflow occurs. It would not be worth the trouble to search a 128 word array just to find the particular x_i or y_i that was causing the problem. In practice, the first loop will work almost all the time, and the second will practically never be needed. The version of CABS traditionally used by Cray essentially uses the method of the second loop all the time. (This example ignores underflows and NaNs for simplicity.)

In summary, the Cray T90 series with IEEE floating point hardware architecture has various means of handling conditions within vector loops, including vector merge, gather/scatter, and vector compress/expand. The worst cases can be handled by restructuring the code and inspecting the exception bits occasionally. The presence of IEEE special cases need not burden a code excessively as long as they occur rarely, or can be handled via vector merge substitutions.

7 Conclusion

The Cray T90 system with IEEE floating point hardware provides a virtually complete implementation of IEEE standard floating-point arithmetic in a parallel-vector architecture. With appropriate compiler and library support, the most useful features of IEEE arithmetic are accessible to the user at minimal performance penalty. The currently known anomalies of Cray arithmetic are eliminated, while maintaining supercomputer-level scalar, vector, and parallel performance. Applications which currently run on Cray PVP vector machines should continue to vectorize well, and will benefit from the better behavior of IEEE arithmetic, such as improved rounding of results. Newer codes which wish to exploit the additional features of IEEE arithmetic such as exception detection and manipulation of ∞ and NaN, may do so. As the examples given here illustrate, in many cases these features of IEEE can be used with little performance penalty.

References

[1] Kiernan, J., 1992. Cray Research and IEEE Floating-point Arithmetic. CUG Spring *Proceedings*, Berlin,53-58.

[2] IEEE 1987. IEEE Standard 754-1985 for binary floating-point arithmetic. IEEE Reprinted in *Sig-Plan* 22,2,9-25.