

“UnInit”

Locating use of uninitialised data in floating point computation in big applications

Oct 31, 2012

NASA Advanced Supercomputing Division

Location of software on Pleiades



/u/scicon/tools/share/uninit

bin

doc

README_uninit_background.txt

README_uninit_methodology.txt

README_uninit_usage.txt

examples

src



Why do we care?

- Pulling in data from uninitialised data is a notorious source of error for applications
- Different run-time environments can result in different values being pulled in, giving different results
- Most visible when changing systems or compilers
- Typically dismissed out-of-hand by users as a potential source of error
- Has a corrosive effect via the requirement to maintain legacy software/environments to preserve the so-called “correct” functioning of some application
- Which of course can stop “working” at any moment
- Users either reticent or ill-equipped to fix this



Some compiler modules on Pleiades

```
comp/intel/10.0.023_64      math/intel_mkl_32_10.0.011
comp/intel/10.0.026_32      math/intel_mkl_64_10.0.011
comp/intel/10.0.026_64      mathematica/7.0.1
comp/intel/10.1.008_32      matlab/2009b
comp/intel/10.1.008_64      matlab/2010b
comp/intel/10.1.011_32      memoryscape/3.2.2-1
comp/intel/10.1.011_64      metis/4.0.1
comp/intel/10.1.013_32      mpfr/2.4.2
comp/intel/10.1.013_64      mpi/mpt.1.25
comp/intel/10.1.015_32      mpi-intel/2011.0
comp/intel/10.1.015_64      mpi-intel/3.1.038
comp/intel/10.1.021_32      mpi-intel/3.1b
comp/intel/10.1.021_64      mpi-intel/3.2.011
comp/intel/11.0.069_32      mpi-intel/4.0.028(default:4.0:4.0.0)
comp/intel/11.0.069_64      mpi-intel/4.0.2.003(4.0.2)
comp/intel/11.0.074_32      mpi-mvapich2/1.2pl/gcc
comp/intel/11.0.074_64      mpi-mvapich2/1.2pl/intel
comp/intel/11.0.081_32      mpi-mvapich2/1.2pl/intel-PIC
comp/intel/11.0.081_64      mpi-mvapich2/1.4.1/gcc
comp/intel/11.0.083_32      mpi-mvapich2/1.4.1/intel
comp/intel/11.0.083_64      mpi-mvapich2/1.6/gcc
comp-intel/11.1.038         mpi-mvapich2/1.6/intel
comp-intel/11.1.046         mpi-sgi/mpt.1.26
comp-intel/11.1.056         mpi-sgi/mpt.2.01
comp-intel/11.1.072         mpi-sgi/mpt.2.04
comp-intel/2011.2(default:2011)  mpi-sgi/mpt.2.04-fsa
comp-intel/2011.4.191(2011.4)  mpi-sgi/mpt.2.04.10789
comp-intel/2011.7.256       nas
comp-pgi/10.6               ncarg/4.4.2/intel
comp-pgi/11.0               ncl/5.1.1
comp-pgi/11.6               ncl/5.2.1
comp-pgi/12.3               ncl/5.2.1.gcc432
```

Intel: the performance compiler of choice, hands down



- Sure would like to get rid of those old compilers
- Users will complain: My code only works with version such-and-such. 99% of the time, sign of problem with their code
- Need to be pulled into newer versions
- Can't really support the older compilers

Typical flags users use to check their code:



- Two popular flags:
- -check
- -ftrapuv

Asserted here:

-check and -ftrapuv are of no use to find uninitialised data. For this, they are less than useless, as they give a false sense of correctness.



-check

Consider the code in "ex1.f":

```
1      program main
2      implicit none
3      double precision d
4      print *, d
5      print 100, d
6      100 format (z)
7      end
```

Lines 4 and 5 print a double precision variable, "d", which has been correctly declared, but never set. When compiled and run, we get the following results:

```
+ ifort ex1.f
+ ./a.out
  0.0000000000000000E+000
      0
```

In this example, the variable happened to contain zero.



-check

If we add the "-check" option, the intel compiler detects at runtime that the variable has not been initialized, and gives a fatal error:

```
+ ifort -check ex1.f  
+ ./a.out
```

```
forrtl: severe (193): Run-Time Check Failure. The variable 'main_$D' is  
being used without being defined
```

Image	PC	Routine	Line	Source
a.out	000000000046970A	Unknown	Unknown	Unknown
a.out	0000000000468285	Unknown	Unknown	Unknown
a.out	000000000041FE66	Unknown	Unknown	Unknown
a.out	0000000000403FA5	Unknown	Unknown	Unknown
a.out	0000000000404A58	Unknown	Unknown	Unknown
a.out	0000000000402CE7	Unknown	Unknown	Unknown
a.out	0000000000402C7C	Unknown	Unknown	Unknown
libc.so.6	00007FFFED133BC6	Unknown	Unknown	Unknown
a.out	0000000000402B79	Unknown	Unknown	Unknown

-check



However, it is easy to inhibit the proper functioning of runtime detection of uninitialized variables by simply passing the variable to a subroutine before it is used. We change our example to the contents of "ex2.f":

```
1      program main
2      implicit none
3      double precision d
4      call toto (d)
5      print *, d
6      print 100, d
7      100 format (z)
8      end
9      subroutine toto (d)
10     implicit none
11     double precision d
12     print *, d
13     print 100, d
14     100 format (z)
15     return
16     end
```

The code is essentially the same as "ex1.f". The difference is, that on line 4, the variable "d" is passed to subroutine "toto". The variable is not set in the subroutine. Simply referencing an otherwise un-set variable in a subroutine argument list looks to be enough to defeat the detection of the fact that variable "d" is uninitialized:

```
+ ifort -check ex2.f
+ ./a.out
0.0000000000000000E+000
      0
0.0000000000000000E+000
      0
```

Again, the variable happened to contain a zero, but this time, the runtime detection of the uninitialized variable did not occur.



-check

Let's try again, turning on full optimization and inter-procedural analysis ("-ipo"), which should have no trouble functioning since the source for the main program and the subroutine are in the same source file "ex2.f", after all:

```
+ ifort -ipo -g -traceback -O3 -check ex2.f
ipo: remark #11001: performing single-file optimizations
ipo: remark #11006: generating object file /tmp/ipo_ifort7cxuTK.o
+ ./a.out
  0.0000000000000000E+000
      0
  0.0000000000000000E+000
      0
```

Still, placing the variable "d" on the argument list of a subroutine entirely defeats the functioning of the "-check" option.



-check

CONCLUSION:

While "-check" may detect some uses of uninitialized variables, it does not detect them all. So, the fact that a code runs without error with "-check" enabled, is no assurance that uninitialized variables don't exist in a user's code.

Still useful for bounds-checking, though!



-ftrapuv

Please see:

`http://software.intel.com/en-us/articles/dont-optimize-when-using-ftrapuv-for-uninitialized-variable-detection`

for an explanation by Intel of why no optimization should be used when compiling with "-ftrapuv". All examples in this document will always use an explicit "-O0" when compiling with "-ftrapuv".

Note that default optimization is "-O2".



-ftrapuv

The "-ftrapuv" option is popular and misunderstood. Perhaps due to its unfortunate name, there is a general belief that this will cause uninitialized floating point variables to generate floating exceptions when they are used. This is not the case. What the option does, is cause some uninitialized variables to be filled with a "large value". In this case, all hexadecimal "C". i.e. a 4 byte entity will be filled with 0xcccccccc, while an 8 byte entity will be filled with 0xcccccccccccccccc.

This is a perfectly valid floating point number, in either 4 or 8 byte (i.e. single or double precision) modes.



-ftrapuv

Let's try this option with our two previous examples:

```
+ ifort -g -traceback -O0 -ftrapuv -fpe0 ex1.f
+ ./a.out
-9.255963134931783E+061
CCCCCCCCCCCCCCCCCC
+ ifort -g -traceback -O0 -ftrapuv -fpe0 ex2.f
+ ./a.out
-9.255963134931783E+061
CCCCCCCCCCCCCCCCCC
-9.255963134931783E+061
CCCCCCCCCCCCCCCCCC
```

Note that we have explicitly enabled floating point exceptions by use of the "-fpe0" option.

So, the result is that "hexadecimal all C" in double precision is approximately $-9.256 \text{ E } 61$. The corresponding value for single precision is left as an exercise for the user.

-ftrapuv



Just to be completely sure about the acceptability of hexadecimal all c's as a floating point value, let's alter "ex2.f" to multiply the uninitialized value of "d" by 2.0 prior to printing it out. The result is "ex2a.f":

```
1      program main
2      implicit none
3      double precision d
4      call toto (d)
5      d = d * 2.0
6      print *, d
7      print 100, d
8      100 format (z)
9      end
10     subroutine toto (d)
11     implicit none
12     double precision d
13     print *, d
14     print 100, d
15     100 format (z)
16     return
17     end
```

same as before, except that the value of "d" is multiplied by 2.0 at line 5.

The result is:

```
+ ifort -g -traceback -O0 -ftrapuv -fpe0 ex2a.f
+ ./a.out
-9.255963134931783E+061
  CCCCCCCCCCCCCC
-1.851192626986357E+062
  CCDCCCCCCCCCCC
```

So, the print of "d" from line 6 is, in fact, twice that from line 13.

Just to clarify how useless the "-ftrapuv" option is, here's one last example "ex2b.f":

```
1      program main
2      implicit none
3      double precision d, e
4      e = 1.0D100
5      call toto (d)
6      e = e + d
7      print *, e
8      end
9      subroutine toto (d)
10     implicit none
11     double precision d
12     d = d * 2.0
13     return
14     end
```

We now compile and run this code with no options, with `-check`, and with `-ftrapuv`

```
+ ifort ex2b.f
+ ./a.out
1.0000000000000000E+100
+ ifort -check ex2b.f
+ ./a.out
1.0000000000000000E+100
+ ifort -g -traceback -O0 -ftrapuv -fpe0 ex2b.f
+ ./a.out
1.0000000000000000E+100
```

We can see from the above results the uselessness of `-check` and `-ftrapuv`, as the output is the same in all modes. The dangerous addition of uninitialized data at line 6 lurks as a potential error.

Even when the `-ftrapuv` flag is used to set the uninitialized data to the value `0xc0000000`, and then multiplied by two at line 12, the addition of the result: `-1.851192626986357E+062` to `1.0D100` makes no difference to the result due to the large difference of exponents.



Conclusion:

`"-check"` is of no use to find uninitialized floating point variables.

`"-ftrapuv"` is of no use to find uninitialized floating point variables.

memory



Consider the following fortran code contained in "ex3.f":

```
1      program main
2      implicit none
3      double precision aa (1000), bb (1000)
4      common /toto/ aa, bb
5      integer n
6      n = 1000
7      call sub1 (n)
8      end
9      subroutine sub1 (n)
10     implicit none
11     integer n
12     double precision a1
13     double precision a2
14     double precision a (n)
15     double precision b (n)
16     double precision c (1000)
17     double precision d (1000)
18     real *8, dimension (:), allocatable :: e, f, g, h
19     double precision aa (1000), bb (1000)
20     common /toto/ aa, bb
21     print *, "sub1 -----"
22     allocate (e (n))
23     allocate (f (n))
24     allocate (g (1000))
25     allocate (h (1000))
26     print 200, "address of sub1 automatic array a", loc (a)
27     print 100, a (1)
28     print 100, a (1000)
29     print 200, "address of sub1 automatic array b", loc (b)
30     print 100, b (1)
31     print 100, b (1000)
32     print 200, "address of sub1 fixed size array c", loc (c)
33     print 100, c (1)
34     print 100, c (1000)
```

memory



```
35     print 200, "address of sub1 fixed size array d", loc (d)
36     print 100, d (1)
37     print 100, d (1000)
38     print 200, "address of sub1 allocatable array e", loc (e)
39     print 100, e (1)
40     print 100, e (1000)
41     print 200, "address of sub1 allocatable array f", loc (f)
42     print 100, f (1)
43     print 100, f (1000)
44     print 200, "address of sub1 allocatable array g", loc (g)
45     print 100, g (1)
46     print 100, g (1000)
47     print 200, "address of sub1 allocatable array h", loc (h)
48     print 100, h (1)
49     print 100, h (1000)
50     print 200, "address of sub1 scalar a1", loc (a1)
51     print 100, a1
52     print 200, "address of sub1 scalar a2", loc (a2)
53     print 100, a2
54     deallocate (e)
55     deallocate (f)
56     deallocate (g)
57     deallocate (h)
58     call mallocator ("sub1 first", 1000)
59     call mallocator ("sub1 second", 1000)
60     print 200, "address of sub1 common-block array aa", loc (aa)
61     print 100, aa (1)
62     print 100, aa (1000)
63     print 200, "address of sub1 common-block array bb", loc (bb)
64     print 100, bb (1)
65     print 100, bb (1000)
66     return
67 100 format (z16)
68 200 format (a, z16)
69     end
```



memory

The purpose of this code is to demonstrate the fundamental ways that variables can be declared in fortran, and discuss the ramifications of how these different declarations affect where the data is actually located by the ifort runtime. Understanding and controlling this is fundamental to developing a methodology for detecting uninitialized variables in user code.

Lines 12 and 13 show the simplest variable of all. A scalar entity declared with a type.

Lines 16 and 17 show a simple array, declared with a fixed size.

Lines 14 and 15 show a so-called "automatic" array. These arrays are declared with a size determined by a variable ("n"), that is itself passed in the argument list of the subroutine. The compiler thus has to manage allocating and de-allocating the space for such arrays upon entry/exit.

Line 18 shows allocatable arrays. These arrays are explicitly allocated and deallocated by the user with code on lines 22-25, and 54-57.

Lines 19-20 show arrays that are contained in common blocks.



memory

Lines 58 and 59 call a small scrap of C code, that allocates memory using a call to "malloc". That code is in "allocator.c":

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  allocator_ (char *s, int *n)
4  {
5  void *p;
6      p = malloc (*n*8);
7      if (p == NULL) {
8          fprintf (stderr, "could not malloc: %d bytes", *n*8);
9          exit (1);
10     }
11     printf ("%s: malloced 8*%d bytes at address: %16llx %16llx\n", s, *n, p,
*(unsigned long long *)p);
12 }
```



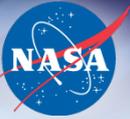
memory

None of these different types of allocated memory are initialized in any way. Generally, their addresses (i.e. "loc") and contents are printed by the running of the resulting compiled program. We are going to compile and run this code in a variety of ways and examine the printed addresses and contents of the data. Our goal is to somehow fill up the uninitialized areas with signaling NANs. i.e. special bit patterns that, when used in a floating point computation, will cause a "floating point exception". This can be used in conjunction with the debugger, i.e. "gdb" or "idbc" to find the location in the user source code where the exception occurred, so that the user can be made aware of when they are using uninitialized floating point data.

It is recommended to use the command line intel debugger, "idbc" for getting the line number information. Since the described method is explicitly only functional with the intel compiler suite, access to the idbc debugger can be assumed. Also, using the command-line version makes incorporation of the idbc command very easy for PBS batch scripts. Lastly, gdb seems to have trouble correctly reading certain Intel-generated symbol tables, particularly for extremely large and complex codes. So, the use of "idbc" is highly recommended.

NOTE that there is no similar concept of an SNAN for integer data or operations.

memory



We will now compile and run "ex3.f" with `-O0`. The results, with line numbers for discussion, are:

```
1 + ifort -O0 ex3.f mallocator.o
2 + ./a.out
3 sub1 -----
4 address of sub1 automatic array a      7FFFFFFF9D00
5      0
6      0
7 address of sub1 automatic array b      7FFFFFFFBC40
8      0
9      2D
10 address of sub1 fixed size array c      6A7DA0
11      0
12      0
13 address of sub1 fixed size array d      6A9EA0
14      0
15      0
16 address of sub1 allocatable array e      6BA190
17      0
18      0
19 address of sub1 allocatable array f      6BC0E0
20      0
21      0
22 address of sub1 allocatable array g      6BE030
23      0
24      0
25 address of sub1 allocatable array h      6BFF80
26      0
27      0
28 address of sub1 scalar a1      7FFFFFFFDF10
29      6B7038
30 address of sub1 scalar a2      7FFFFFFFDF28
31      7FFFEADABF6F8
32 sub1 first: malloced 8*1000 bytes at address:      6ba190      7fffed46eeb8
33 sub1 second: malloced 8*1000 bytes at address:      6bc0e0      0
34 address of sub1 common-block array aa      6B2A80
35      0
36      0
37 address of sub1 common-block array bb      6B49C0
38      0
39      0
```



memory

Note the appearance of "garbage", i.e. unexpected non-zero values, at lines 9, 29, 31, and 32. If you run this code yourself, you may get different values.

Note that the addresses of allocatable arrays e and f on lines 16 and 19 have been re-used by malloc on lines 32 and 33. So we can assume that the ifort allocatable mechanism is using malloc for allocation.

Now, we are going to compile the code with some extra flags. We've determined

that adding "-auto" and "-heap-arrays" has some particular benefits for what

we are trying to accomplish. The "-auto" flag will place all scalars on the stack, while "-heap-arrays" will cause all arrays to be allocated via malloc.

Why we want to do this will become apparent later. We are also going to add "-g -traceback" to get symbol information, "-ftrapuv" to try to set uninitialized data to "all C's", and "-O0" is required with "-ftrapuv".

memory



The results are:

```
1      + ifort -g -traceback -O0 -ftrapuv -fpe0 -auto -heap-arrays ex3.f mallocator.o
2      + ./a.out
3      subl -----
4      address of subl automatic array a          6B4FB0
5              0
6              0
7      address of subl automatic array b          6B3060
8              0
9              0
10     address of subl fixed size array c        7FFFFFFF9C20
11     CCCCCCCCCCCCCC
12     CCCCCCCCCCCCCC
13     address of subl fixed size array d        7FFFFFFFB60
14     CCCCCCCCCCCCCC
15     CCCCCCCCCCCCCC
16     address of subl allocatable array e        6BA030
17              0
18              0
19     address of subl allocatable array f        6BBF80
20              0
21              0
22     address of subl allocatable array g        6BDED0
23              0
24              0
25     address of subl allocatable array h        6BFE20
26              0
27              0
28     address of subl scalar a1          7FFFFFFFD00
29     CCCCCCCCCCCCCC
30     address of subl scalar a2          7FFFFFFFD18
31     CCCCCCCCCCCCCC
32     subl first: malloced 8*1000 bytes at address:          6ba030          7fffed26aeb8
33     subl second: malloced 8*1000 bytes at address:          6bbf80          0
34     address of subl common-block array aa        6AE860
35              0
36              0
37     address of subl common-block array bb        6B07A0
38              0
39              0
```



memory

We can see the severe limits of "-ftrapuv"'s ability to set uninitialized data to "all C's". In fact, only fixed size arrays and scalars got set.

What is helpful, though, is that judging by the range of their addresses, all the other data arrays have been moved to the heap (i.e. malloc).

What can be done now, is to write a little pre-loadable shared code segment that can intercept calls to malloc, and initialize the data before the call returns to the user. That code is not presented here (it's about 200 lines of c code).

memory



However, the results of using it are:

```
1 + LD_PRELOAD=/u/scicon/tools/share/uninsnan_preload.so
2 + ./a.out
3 snan.c line 110: SNAN v1.3 enabled, mode is default: SNAN_MODE_DP, malloc'ed memory will be set to
0xffff7fffffff
4 snan.c line 111: SNAN available modes are: setenv SNAN_MODE_MIXED, setenv SNAN_MODE_DP, setenv
SNAN_MODE_SP, setenv SNAN_MODE_BIGINT, setenv SNAN_MODE_ZEROS
5 snan.c line 112: SNAN works best with ifort and the -heap-arrays -fpe0 flags
6 snan.c line 113: SNAN also works with pgf90 and the -Ktrap=fp flag
7 snan.c line 115: SNAN will abort if more than one openmp thread is used, over-ride with: setenv
SNAN_MODE_MANY_THREADS
8 snan.c line 119: SNAN messages may be inhibited with: setenv SNAN_MODE_QUIET
9 subl -----
10 address of subl automatic array a          6B4FB0
11 FFF7FFFFFFFF
12 FFF7FFFFFFFF
13 address of subl automatic array b          6B3060
14 FFF7FFFFFFFF
15 FFF7FFFFFFFF
16 address of subl fixed size array c        7FFFFFFF9C20
17 CCCCCCCCCCCCCC
18 CCCCCCCCCCCCCC
19 address of subl fixed size array d        7FFFFFFFB60
20 CCCCCCCCCCCCCC
21 CCCCCCCCCCCCCC
22 address of subl allocatable array e       6B9020
23 FFF7FFFFFFFF
24 FFF7FFFFFFFF
25 address of subl allocatable array f       6BAF70
26 FFF7FFFFFFFF
27 FFF7FFFFFFFF
28 address of subl allocatable array g       6BCEC0
29 FFF7FFFFFFFF
30 FFF7FFFFFFFF
31 address of subl allocatable array h       6BEE10
32 FFF7FFFFFFFF
33 FFF7FFFFFFFF
34 address of subl scalar a1                 7FFFFFFFD00
35 CCCCCCCCCCCCCC
36 address of subl scalar a2                 7FFFFFFFD18
37 CCCCCCCCCCCCCC
38 address of subl common-block array aa     6AE860
39 0
40 0
41 address of subl common-block array bb     6B07A0
42 0
43 0
44 subl first: malloced 8*1000 bytes at address:      6b9020 fff7fffffff
45 subl second: malloced 8*1000 bytes at address:     6baf70 fff7fffffff
```



memory

Note that explicitly setting `LD_PRELOAD` from the command line is quite dangerous, as it applies to all further commands executed by the user's login shell. From now on in the documentation and examples, we will use the "snan_wrapper" script to perform this function. This is a script which takes the target command as an argument, and only applies the setting of `LD_PRELOAD` within the script, while the target command is being run. This prevents confusion and possible error for the user due to inadvertent setting of `LD_PRELOAD`.

So, now we are approaching our goal. We are able to initialize all the basic types of fortran memory allocation except common blocks. Traditional common blocks live in "bss" space, and can be assumed by the user to be preset to zero. More modern allocatable common blocks will use the allocate mechanism, and be subject to initialization by the malloc preloaded shared object method.

How are we to deal with those "all C" regions?

The answer is fairly simple. The compiler actually generates code that appears in the user's executable to explicitly set these regions at runtime. It is a simple matter to create a tool which will directly edit the executable created by the compiler, replacing all the instruction sequences that set memory areas to "all C" with a constant of our choice. In this case, a signaling NaN.

memory



This process appears as follows:

```
1 + ifort -g -traceback -O0 -ftrapuv -fpe0 -auto -heap-arrays ex3.f mallocators.o
2 + snan_patch -i a.out -o a.out.alt
3 snan_patch: 6 movl opcodes had constant changed from 0xcccccccc to 0xffff7fff (signalling NAN, double
precision)
4 + snan_wrapper ./a.out.alt
5 snan.c line 231: SNAN v1.3 enabled, mode is default: SNAN_MODE_DP, malloc'ed memory will be set to
0xffff7ffffffffffffll
6 snan.c line 235: SNAN available modes are: setenv SNAN_MODE_MIXED, setenv SNAN_MODE_DP, setenv
SNAN_MODE_SP, setenv SNAN_MODE_BIGINT, setenv SNAN_MODE_ZEROS
7 snan.c line 236: SNAN works best with ifort and the -heap-arrays -fpe0 flags
8 snan.c line 237: SNAN also works with pgf90 and the -Ktrap=fp flag
9 snan.c line 239: SNAN will abort if more than one openmp or posix thread is used, over-ride with: setenv
SNAN_MODE_MANY_THREADS
10 snan.c line 243: SNAN messages may be inhibited with: setenv SNAN_MODE_QUIET
11 subl -----
12 address of subl automatic array a          6B4FB0
13 FFF7FFFFFFFFFFFFFFF
14 FFF7FFFFFFFFFFFFFFF
15 address of subl automatic array b          6B3060
16 FFF7FFFFFFFFFFFFFFF
17 FFF7FFFFFFFFFFFFFFF
18 address of subl fixed size array c          7FFFFFFF9C20
19 FFF7FFFFFFFF7FFF
20 FFF7FFFFFFFF7FFF
21 address of subl fixed size array d          7FFFFFFFBB60
22 FFF7FFFFFFFF7FFF
23 FFF7FFFFFFFF7FFF
24 address of subl allocatable array e          6B9020
25 FFF7FFFFFFFFFFFFFFF
26 FFF7FFFFFFFFFFFFFFF
27 address of subl allocatable array f          6BAF70
28 FFF7FFFFFFFFFFFFFFF
29 FFF7FFFFFFFFFFFFFFF
30 address of subl allocatable array g          6BCEC0
31 FFF7FFFFFFFFFFFFFFF
32 FFF7FFFFFFFFFFFFFFF
33 address of subl allocatable array h          6BEE10
34 FFF7FFFFFFFFFFFFFFF
35 FFF7FFFFFFFFFFFFFFF
36 address of subl scalar a1          7FFFFFFFDF00
37 FFF7FFFFFFFF7FFF
38 address of subl scalar a2          7FFFFFFFDF18
39 FFF7FFFFFFFF7FFF
40 address of subl common-block array aa          6AE860
41 0
42 0
43 address of subl common-block array bb          6B07A0
44 0
45 0
46 subl first: malloced 8*1000 bytes at address:          6b9020 fff7ffffffffffff
47 subl second: malloced 8*1000 bytes at address:          6baf70 fff7ffffffffffff
```



memory

Line 1 builds `ex3.f` with the same set of flags as before.

Line 2 shows the application of the `"snan_patch"` utility. Input is the `a.out` just created, and output is `"a.aout.alt"`. By default, the "all Cs" are set to signaling NaN double precision.

Line 3 is output from `snan_patch`, showing that 6 opcodes were changed.

Line 4 uses `"snan_wrapper"` to run `"a.out.alt"`. `snan_wrapper` is a simple script that runs the indicated program while setting `LD_PRELOAD` internally, only for the running of the program, and not affecting your shell environment. This is the preferable way to set `LD_PRELOAD`. Please do not set `LD_PRELOAD` yourself.

Lines 11 - 47 show that all uninitialized data has been set to double precision signaling NaN's, with the exception of common blocks, as discussed above.

Note that either `FFF7FFFFFFFFFFFFFF` or `FFF7FFFFFFFF7FFFF` are signalling NaNs in double precision. Since the compiler only uses 4 byte stores for setting "all Cs", we have to repeat the exponent in the mantissa portion. Not a problem, since all that is required is that the mantissa is non-zero, which is satisfied in both cases.



memory

Important note:

How do we know that our code doesn't contain sequences setting data to "all Cs" already?
Isn't there the possibility of "snan_patch" to break our code?

The answer to the first question is: we don't, so yes, `snan_patch` may break the code.
To address this, `snan_patch` has an option to simply look for the code sequences in question.
To use this function, compile WITHOUT `-ftrapuv` as follows:

```
1 + ifort -g -traceback -O0 -fpe0 -auto -heap-arrays ex3.f malloc.o
2 + snan_patch -v a.out
3 main.c line 338: no movl opcodes of the constant 0xccccccc to a stack based address were detected, OK to rebuild
with -ftrapuv and apply snan_patch
```

On line 1, we compile the code with all the usual options EXCEPT `-ftrapuv`.

Line 2 uses `snan_patch` with the `-v` option to examine `a.out` for the code sequences.

Line 3 shows that no such sequences were detected, therefore, it is safe to add the `-ftrapuv` flag, RECOMPILE, and use `snan_patch` to create an alternate `a.out`.

When "snan_patch" is used with the "-v" option, it will return a non-zero exit status if the target `movl` opcodes are found in the input executable. This is specifically so that "snan_patch -v" can be used inside a Makefile and the error condition of target `movl` opcodes present in an executable compiled without `-ftrapuv` can be caught.

Please use appropriately.



usage

HOW TO FIND UNINITIALIZED FLOATING POINT VARIABLES USING ifort

Step 1) Change your ifort code generation flags to use:

```
-O0 -ftrapuv -heap-arrays -auto -g -traceback -fpe0
```

If your code does not compile/run with this set of options, then your code cannot use the method described here. You must use precisely these flags except as noted here:

Other flags known to work with the above set:

```
-fPIC  
-fp-model precise
```

You MUST compile AND RELINK ALL of your object modules (.o .a .so files) and RELINK with the above flags.

(actually, any flags not affecting code generation are OK)



usage

Step 2) Put:

`/u/scicon/tools/share/uninit/bin`

in your PATH.

There's a script, an executable, and a shared loadable library there. The source of the "snan_patch" executable is in:

`/u/scicon/tools/share/uninit/src`

The source of the "snan_preload.so" shared object is in:

`/u/scicon/tools/share/uninit/src`



usage

Step 3) Apply the "snan_patch" command to your executable

For double precision code:

```
snan_patch -D -i a.out -o a.out.altd
```

For single precision code:

```
snan_patch -S -i a.out -o a.out.alts
```

If neither -S nor -D are supplied, the default is double precision (-D)

If you are unsure if your code is double or single precision, simply make both the single and double precision versions as shown above, and try running each one as described below.

Be sure to retain the unaltered "a.out" version of your code.

snan_patch must always use as input the original a.out as created in Step 1.



usage

Step 4) Run your code using the `snan_wrapper` script

To ensure a complete core file, add this command before your run:

```
limit coredumpsize unlimited
```

now:

For double precision:

```
setenv SNAN_MODE_DP  
snan_wrapper a.out.dp
```

For single precision:

```
setenv SNAN_MODE_SP  
snan_wrapper a.out.dp
```

If your code tried to perform floating point computation with uninitialized data, a floating point exception will be generated, and a core dump created.

If no floating point exception occurs, then you are not computing using uninitialized data. To verify the integrity of the process, you could deliberately add such code to your program, to ensure that the detection process is working correctly, and that you have correctly followed the steps outlined above.



usage

Step 5) Examining the core file if one is created.

Invoke a debugger to get the line information where the floating point exception occurred. You can then fix your code, and iterate on this process until you have located and repaired all locations in your code where uninitialized data was being used in computation.

invoke `idbc`, the intel command-line debugger:

```
idbc ./a.out core.nnnn
```

when a core dump is created, and "core.nnn" is the name of the core file.

Use the "where" command to find where the floating point exception occurred. That is the location of the use of the uninitialized data.

You may equally well be able to use the `gnu gdb` debugger instead of `idbc`. `gdb` sometimes has trouble reading the `ifort` symbol tables. That is why `idbc` is suggested in it's place.



usage

OPENMP IMPORTANT NOTE-----

The shared library will disable the ability to run with more than one thread.

Using this method, you must debug your openmp code with:

```
setenv OMP_NUM_THREADS 1
```

The ifort runtime cannot successfully process floating point exceptions from more than one thread.

In such a case, you will receive meaningless and misleading error messages or no messages at all.

NOTE-----

Some serial codes call library routines that perform certain functions via the pthreads mechanism. In such a case, you can over-ride the shared library's inhibition of threading by using an environment variable:

```
setenv SNAN_MODE_MANY_THREADS
```



usage

File `ex5.f` contains:

```
1      program main
2      implicit none
3      double precision d
4      d = d + 1.0
5      print *, d
6      print 100, d
7      100 format (z)
8      end
```

usage



```
1 limit coredumpsize unlimited
2 rm -f core.19851
3 ifort -O0 -ftrapuv -fpe0 -auto -heap-arrays -g -traceback ex5.f
4 snan_patch -i a.out -o a.out.alt
5 snan_patch: 3 movl opcodes had constant changed from 0xcccccccc to 0xffff7ffff (signalling NAN, double precision)
6 snan_wrapper ./a.out.alt
7 snan.c line 231: SNAN v1.3 enabled, mode is default: SNAN_MODE_DP, malloc'ed memory will be set to 0xffff7ffffffffffLL
8 snan.c line 235: SNAN available modes are: setenv SNAN_MODE_MIXED, setenv SNAN_MODE_DP, setenv SNAN_MODE_SP, setenv
SNAN_MODE_BIGINT, setenv SNAN_MODE_ZEROS
9 snan.c line 236: SNAN works best with ifort and the -heap-arrays -fpe0 flags
10 snan.c line 237: SNAN also works with pgf90 and the -Ktrap=fp flag
11 snan.c line 239: SNAN will abort if more than one openmp or posix thread is used, over-ride with: setenv
SNAN_MODE_MANY_THREADS
12 snan.c line 243: SNAN messages may be inhibited with: setenv SNAN_MODE_QUIET
13 forrtl: error (65): floating invalid
14 Image                PC                Routine                Line                Source
15 a.out.alt             0000000000402D01  MAIN__                4                   ex5.f
16 a.out.alt             0000000000402C7C  Unknown              Unknown             Unknown
17 libc.so.6             00007FFFECD2CBC6  Unknown              Unknown             Unknown
18 a.out.alt             0000000000402B79  Unknown              Unknown             Unknown
19 snan_wrapper: line 3: 25728 Aborted
20 setenv LASTCORE `ls -t core* | sed 1q`
21 sed 1q
22 ls -t core.25728
23 idbc ./a.out.alt core.25728
24 Intel(R) Debugger for applications running on Intel(R) 64, Version 12.0, Build [74.923.2]
25 -----
26 object file name: ./a.out.alt
27 core file name: core.25728
28 Reading symbols from /home4/dpbarker/uninit/examples/a.out.alt...done.
29 Core file produced from executable a.out.alt
30 Initial part of arglist: ./a.out.alt
31 Thread terminated at PC 0x00007fffedc40945 by signal SIGABRT
32 #0 0x00007fffedc40945 in raise () in /lib64/libc-2.11.1.so
33 #1 0x00007fffedc41f21 in abort () in /lib64/libc-2.11.1.so
34 #2 0x000000000040334e in for_signal_handler () in /home4/dpbarker/uninit/examples/a.out.alt
35 #3 0x00007fffed27f5d0 in __restore_rt () in /lib64/libpthread-2.11.1.so
36 #4 0x0000000000402d01 in main () at /home4/dpbarker/uninit/examples/ex5.f:4
```



usage

ex7.c consists of:

```
1  #include <stdio.h>
2  main ()
3  {
4  double d;
5      yoyo (&d);
6      toto (&d);
7      printf ("%lg\n", d);
8  }
9  toto (double *d)
10 {
11     *d = *d + 1.0;
12 }
13 yoyo (long long *d)
14 {
15     printf ("%llx\n", *d);
16 }
```



usage

```
1 limit coredumpsize unlimited
2 rm -f core.5012
3 icc -g -traceback -OO -ftrapuv -heap-arrays ex7.c
4 ./a.out
5 cccccccccccccccc
6 -9.25596e+61
7 icc -g -traceback -OO -ftrapuv -heap-arrays ex7.c
8 snan_patch -i a.out -o a.out.alt
9 snan_patch: 9 movl opcodes had constant changed from 0xcccccccc to 0xffff7ffff (signalling NAN,
double precision)
10 snan_wrapper a.out.alt
11 snan.c line 231: SNAN v1.3 enabled, mode is default: SNAN_MODE_DP, malloc'ed memory will be set
to 0xffff7ffffffffffLL
12 snan.c line 235: SNAN available modes are: setenv SNAN_MODE_MIXED, setenv SNAN_MODE_DP, setenv
SNAN_MODE_SP, setenv SNAN_MODE_BIGINT, setenv SNAN_MODE_ZEROS
13 snan.c line 236: SNAN works best with ifort and the -heap-arrays -fpe0 flags
14 snan.c line 237: SNAN also works with pgf90 and the -Ktrap=fp flag
15 snan.c line 239: SNAN will abort if more than one openmp or posix thread is used, over-ride with:
setenv SNAN_MODE_MANY_THREADS
16 snan.c line 243: SNAN messages may be inhibited with: setenv SNAN_MODE_QUIET
17 snan_wrapper: line 3: 5048 Floating point exception(core dumped) LD_PRELOAD=${HERE}/
snan_preload.so @$
18 setenv LASTCORE `ls -t core* | sed 1q`
19 sed 1q
20 ls -t core.5048
21 idbc ./a.out.alt core.5048
22 Intel(R) Debugger for applications running on Intel(R) 64, Version 12.0, Build [74.923.2]
23 -----
24 object file name: ./a.out.alt
25 core file name: core.5048
26 Reading symbols from /home4/dpbarker/uninit/examples/a.out.alt...done.
27 Core file produced from executable a.out.alt
28 Initial part of arglist: a.out.alt
29 Thread terminated at PC 0x00000000040063d by signal SIGFPE
30 #0 0x00000000040063d in toto (d=0x7ffffffffffe2c0) at /home4/dpbarker/uninit/examples/ex7.c:11
31 #1 0x0000000004005ce in main () at /home4/dpbarker/uninit/examples/ex7.c:6
```



Future plans

- Some small-ish effort to support realloc
- Release the source code to interested parties
- Discuss with Intel/others
- Promote use as part of build/validation for large apps
- Can this be made simpler??



Questions, help:

david.p.barker@nasa.gov
dbarker@supersmith.com