Next: Preprocessor Options, Previous: Optimize Options, Up: Invoking GCC [Contents] [Index]

3.11 Program Instrumentation Options

GCC supports a number of command-line options that control adding run-time instrumentation to the code it normally generates. For example, one purpose of instrumentation is collect profiling statistics for use in finding program hot spots, code coverage analysis, or profile-guided optimizations. Another class of program instrumentation is adding run-time checking to detect programming errors like invalid pointer dereferences or out-of-bounds array accesses, as well as deliberately hostile attacks such as stack smashing or C++ vtable hijacking. There is also a general hook which can be used to implement other forms of tracing or function-level instrumentation for debug or program analysis purposes.

-p

Generate extra code to write profile information suitable for the analysis program prof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

-pg

Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

-fprofile-arcs

Add code so that program flow *arcs* are instrumented. During execution the program records how many times each branch and call is executed and how many times it is taken or returns. On targets that support constructors with priority support, profiling properly handles constructors, destructors and C++ constructors (and destructors) of classes which are used as a type of a global variable.

When the compiled program exits it saves this data to a file called *auxname*.gcda for each source file. The data may be used for profile-directed optimizations (-fbranch-probabilities), or for test coverage analysis (-ftest-coverage). Each object file's *auxname* is generated from the name of the output file, if explicitly specified and it is not the final

executable, otherwise it is the basename of the source file. In both cases any suffix is removed (e.g. foo.gcda for input file dir/foo.c, or dir/foo.gcda for output file specified as -o dir/foo.o). See Cross-profiling.

--coverage

This option is used to compile and link code instrumented for coverage analysis. The option is a synonym for -fprofile-arcs -ftest-coverage (when compiling) and -lgcov (when linking). See the documentation for those options for more details.

- Compile the source files with -fprofile-arcs plus optimization and code generation options. For test coverage analysis, use the additional -ftest-coverage option. You do not need to profile every source file in a program.
- Compile the source files additionally with -fprofile-abs-path to create absolute path names in the .gcno files. This allows gcov to find the correct sources in projects where compilations occur with different working directories.
- Link your object files with -lgcov or -fprofile-arcs (the latter implies the former).
- Run the program on a representative workload to generate the arc profile information. This may be repeated any number of times. You can run concurrent instances of your program, and provided that the file system supports locking, the data files will be correctly updated. Also fork calls are detected and correctly handled (double counting will not happen).
- For profile-directed optimizations, compile the source files again with the same optimization and code generation options plus -fbranch-probabilities (see Options that Control Optimization).
- For test coverage analysis, use gcov to produce human readable information from the .gcno and .gcda files. Refer to the gcov documentation for further information.

With -fprofile-arcs, for each function of your program GCC creates a program flow graph, then finds a spanning tree for the graph. Only arcs that are not on the spanning tree have to be instrumented: the compiler adds code to count the number of times that these arcs are executed. When an arc is the only exit or only entrance to a block, the instrumentation code can be added to the block; otherwise, a new basic block must be created to hold the instrumentation code.

-ftest-coverage

Produce a notes file that the gcov code-coverage utility (see gcov—a Test Coverage Program) can use to show program coverage. Each source file's note file is called

auxname.gcno. Refer to the -fprofile-arcs option above for a description of *auxname* and instructions on how to generate test coverage data. Coverage data matches the source files more closely if you do not optimize.

-fprofile-abs-path

Automatically convert relative source file names to absolute path names in the .gcno files. This allows gcov to find the correct sources in projects where compilations occur with different working directories.

-fprofile-dir=*path*

Set the directory to search for the profile data files in to *path*. This option affects only the profile data generated by -fprofile-generate, -ftest-coverage, -fprofile-arcs and used by -fprofile-use and -fbranch-probabilities and its related options. Both absolute and relative paths can be used. By default, GCC uses the current directory as *path*, thus the profile data file appears in the same directory as the object file.

-fprofile-generate

-fprofile-generate=*path*

Enable options usually used for instrumenting application to produce profile useful for later recompilation with profile feedback based optimization. You must use -fprofile-generate both when compiling and when linking your program.

The following options are enabled: -fprofile-arcs, -fprofile-values, -fvpt.

If *path* is specified, GCC looks at the *path* to find the profile feedback data files. See -fprofile-dir.

To optimize the program based on the collected profile information, use -fprofile-use. See Optimize Options, for more information.

-fprofile-update=*method*

Alter the update method for an application instrumented for profile feedback based optimization. The *method* argument should be one of 'single', 'atomic' or 'prefer-atomic'. The first one is useful for single-threaded applications, while the second one prevents profile corruption by emitting thread-safe code.

Warning: When an application does not properly join all threads (or creates an detached thread), a profile file can be still corrupted.

Using 'prefer-atomic' would be transformed either to 'atomic', when supported by a target, or to 'single' otherwise. The GCC driver automatically selects 'prefer-atomic' when -pthread is present in the command line.

-fsanitize=address

Enable AddressSanitizer, a fast memory error detector. Memory access instructions are instrumented to detect out-of-bounds and use-after-free bugs. The option enables -fsanitize-address-use-after-scope. See https://github.com/google/sanitizers/wiki/AddressSanitizer for more details. The run-time behavior can be influenced using the ASAN_OPTIONS environment variable. When set to help=1, the available options are shown at startup of the instrumented program. See https://github.com/google/sanitizers/wiki/AddressSanitizerFlags#run-time-flags for a list of supported options. The option cannot be combined with -fsanitize=thread and/or -fcheck-pointer-bounds.

-fsanitize=kernel-address

Enable AddressSanitizer for Linux kernel. See https://github.com/google/kasan/wiki for more details. The option cannot be combined with -fcheck-pointer-bounds.

-fsanitize=thread

Enable ThreadSanitizer, a fast data race detector. Memory access instructions are instrumented to detect data race bugs. See https://github.com/google/sanitizers/wiki#threadsanitizer for more details. The run-time behavior can be influenced using the TSAN_OPTIONS environment variable; see https://github.com/google/sanitizers/wiki/ThreadSanitizerFlags for a list of supported options. The option cannot be combined with -fsanitize=address, -fsanitize=leak and/or -fcheck-pointer-bounds.

Note that sanitized atomic builtins cannot throw exceptions when operating on invalid memory addresses with non-call exceptions (-fnon-call-exceptions).

-fsanitize=leak

Enable LeakSanitizer, a memory leak detector. This option only matters for linking of executables and the executable is linked against a library that overrides malloc and other allocator functions. See https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer for more details. The run-time behavior can be influenced using the LSAN_OPTIONS environment variable. The option cannot be combined with -fsanitize=thread.

-fsanitize=undefined

Enable UndefinedBehaviorSanitizer, a fast undefined behavior detector. Various computations are instrumented to detect undefined behavior at runtime. Current suboptions are:

-fsanitize=shift

This option enables checking that the result of a shift operation is not undefined. Note that what exactly is considered undefined differs slightly between C and C++, as well as between ISO C90 and C99, etc. This option has two suboptions, -fsanitize=shift-base and -fsanitize=shift-exponent.

-fsanitize=shift-exponent

This option enables checking that the second argument of a shift operation is not negative and is smaller than the precision of the promoted first argument.

-fsanitize=shift-base

If the second argument of a shift operation is within range, check that the result of a shift operation is not undefined. Note that what exactly is considered undefined differs slightly between C and C++, as well as between ISO C90 and C99, etc.

-fsanitize=integer-divide-by-zero

Detect integer division by zero as well as INT_MIN / -1 division.

-fsanitize=unreachable

With this option, the compiler turns the __builtin_unreachable call into a diagnostics message call instead. When reaching the __builtin_unreachable call, the behavior is undefined.

-fsanitize=vla-bound

This option instructs the compiler to check that the size of a variable length array is positive.

-fsanitize=null

This option enables pointer checking. Particularly, the application built with this option turned on will issue an error message when it tries to dereference a NULL pointer, or if a reference (possibly an rvalue reference) is bound to a NULL

pointer, or if a method is invoked on an object pointed by a NULL pointer.

-fsanitize=return

This option enables return statement checking. Programs built with this option turned on will issue an error message when the end of a non-void function is reached without actually returning a value. This option works in C++ only.

-fsanitize=signed-integer-overflow

This option enables signed integer overflow checking. We check that the result of `+`, `*`, and both unary and binary `-` does not overflow in the signed arithmetics. Note, integer promotion rules must be taken into account. That is, the following is not an overflow:

```
signed char a = SCHAR_MAX;
a++;
```

-fsanitize=bounds

This option enables instrumentation of array bounds. Various out of bounds accesses are detected. Flexible array members, flexible array member-like arrays, and initializers of variables with static storage are not instrumented. The option cannot be combined with `-fcheck-pointer-bounds`.

-fsanitize=bounds-strict

This option enables strict instrumentation of array bounds. Most out of bounds accesses are detected, including flexible array members and flexible array member-like arrays. Initializers of variables with static storage are not instrumented. The option cannot be combined with `-fcheck-pointer-bounds`.

-fsanitize=alignment

This option enables checking of alignment of pointers when they are dereferenced, or when a reference is bound to insufficiently aligned target, or when a method or constructor is invoked on insufficiently aligned object.

-fsanitize=object-size

This option enables instrumentation of memory references using the `__builtin_object_size` function. Various out of bounds pointer accesses are detected.

-fsanitize=float-divide-by-zero

Detect floating-point division by zero. Unlike other similar options, -fsanitize=float-divide-by-zero is not enabled by -fsanitize=undefined, since floating-point division by zero can be a legitimate way of obtaining infinities and NaNs.

-fsanitize=float-cast-overflow

This option enables floating-point type to integer conversion checking. We check that the result of the conversion does not overflow. Unlike other similar options, -fsanitize=float-cast-overflow is not enabled by -fsanitize=undefined. This option does not work well with FE_INVALID exceptions enabled.

-fsanitize=nonnull-attribute

This option enables instrumentation of calls, checking whether null values are not passed to arguments marked as requiring a non-null value by the nonnull function attribute.

-fsanitize=returns-nonnull-attribute

This option enables instrumentation of return statements in functions marked with returns_nonnull function attribute, to detect returning of null values from such functions.

-fsanitize=bool

This option enables instrumentation of loads from bool. If a value other than 0/1 is loaded, a run-time error is issued.

-fsanitize=enum

This option enables instrumentation of loads from an enum type. If a value outside the range of values for the enum type is loaded, a run-time error is issued.

-fsanitize=vptr

This option enables instrumentation of C++ member function calls, member accesses and some conversions between pointers to base and derived classes, to verify the referenced object has the correct dynamic type.

-fsanitize=pointer-overflow

This option enables instrumentation of pointer arithmetics. If the pointer arithmetics overflows, a run-time error is issued.

-fsanitize=builtin

This option enables instrumentation of arguments to selected builtin functions. If an invalid value is passed to such arguments, a run-time error is issued. E.g. passing 0 as the argument to __builtin_ctz or __builtin_clz invokes undefined behavior and is diagnosed by this option.

While -ftrapv causes traps for signed overflows to be emitted, -fsanitize=undefined gives a diagnostic message. This currently works only for the C family of languages.

-fno-sanitize=all

This option disables all previously enabled sanitizers. -fsanitize=all is not allowed, as some sanitizers cannot be used together.

-fasan-shadow-offset=*number*

This option forces GCC to use custom shadow offset in AddressSanitizer checks. It is useful for experimenting with different shadow memory layouts in Kernel AddressSanitizer.

-fsanitize-sections=*s1*,*s2*,...

Sanitize global variables in selected user-defined sections. *si* may contain wildcards.

-fsanitize-recover[=*opts*]

-fsanitize-recover= controls error recovery mode for sanitizers mentioned in comma-separated list of *opts*. Enabling this option for a sanitizer component causes it to attempt to continue running the program as if no error happened. This means multiple runtime errors can be reported in a single program run, and the exit code of the program may indicate success even when errors have been reported. The -fno-sanitize-recover= option can be used to alter this behavior: only the first detected error is reported and program then exits with a non-zero exit code.

Currently this feature only works for -fsanitize=undefined (and its suboptions except for -fsanitize=unreachable and -fsanitize=return), -fsanitize=float-cast-overflow, -fsanitize=float-divide-by-zero, -fsanitize=bounds-strict, -fsanitize=kernel-address and -fsanitize=address. For these sanitizers error recovery is turned on by default, except -fsanitize=address, for which this feature is

experimental. -fsanitize-recover=all and -fno-sanitize-recover=all is also accepted, the former enables recovery for all sanitizers that support it, the latter disables recovery for all sanitizers that support it.

Even if a recovery mode is turned on the compiler side, it needs to be also enabled on the runtime library side, otherwise the failures are still fatal. The runtime library defaults to halt_on_error=0 for ThreadSanitizer and UndefinedBehaviorSanitizer, while default value for AddressSanitizer is halt_on_error=1. This can be overridden through setting the halt_on_error flag in the corresponding environment variable.

Syntax without an explicit *opts* parameter is deprecated. It is equivalent to specifying an *opts* list of:

```
undefined,float-cast-overflow,float-divide-by-zero,bounds-strict
```

-fsanitize-address-use-after-scope

Enable sanitization of local variables to detect use-after-scope bugs. The option sets -fstack-reuse to 'none'.

-fsanitize-undefined-trap-on-error

The -fsanitize-undefined-trap-on-error option instructs the compiler to report undefined behavior using __builtin_trap rather than a libubsan library routine. The advantage of this is that the libubsan library is not needed and is not linked in, so this is usable even in freestanding environments.

-fsanitize-coverage=trace-pc

Enable coverage-guided fuzzing code instrumentation. Inserts a call to __sanitizer_cov_trace_pc into every basic block.

-fsanitize-coverage=trace-cmp

Enable dataflow guided fuzzing code instrumentation. Inserts a call to __sanitizer_cov_trace_cmp1, __sanitizer_cov_trace_cmp2, __sanitizer_cov_trace_cmp4 or __sanitizer_cov_trace_cmp8 for integral comparison with both operands variable or __sanitizer_cov_trace_const_cmp1, __sanitizer_cov_trace_const_cmp2, __sanitizer_cov_trace_const_cmp4 or __sanitizer_cov_trace_const_cmp8 for integral comparison with one operand constant, __sanitizer_cov_trace_cmpf or __sanitizer_cov_trace_cmpd for float or double comparisons and __sanitizer_cov_trace_switch for switch statements.

-fbounds-check

For front ends that support it, generate additional code to check that indices used to access arrays are within the declared range. This is currently only supported by the Fortran front end, where this option defaults to false.

-fcheck-pointer-bounds

Enable Pointer Bounds Checker instrumentation. Each memory reference is instrumented with checks of the pointer used for memory access against bounds associated with that pointer.

Currently there is only an implementation for Intel MPX available, thus x86 GNU/Linux target and -mmpx are required to enable this feature. MPX-based instrumentation requires a runtime library to enable MPX in hardware and handle bounds violation signals. By default when -fcheck-pointer-bounds and -mmpx options are used to link a program, the GCC driver links against the libmpx and libmpxwrappers libraries. Bounds checking on calls to dynamic libraries requires a linker with -z bndplt support; if GCC was configured with a linker without support for this option (including the Gold linker and older versions of ld), a warning is given if you link with -mmpx without also specifying -static, since the overall effectiveness of the bounds checking protection is reduced. See also -static-libmpxwrappers.

MPX-based instrumentation may be used for debugging and also may be included in production code to increase program security. Depending on usage, you may have different requirements for the runtime library. The current version of the MPX runtime library is more oriented for use as a debugging tool. MPX runtime library usage implies -lpthread. See also -static-libmpx. The runtime library behavior can be influenced using various CHKP_RT_* environment variables. See https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler for more details.

Generated instrumentation may be controlled by various -fchkp-* options and by the bnd_variable_size structure field attribute (see Type Attributes) and bnd_legacy, and bnd_instrument function attributes (see Function Attributes). GCC also provides a number of built-in functions for controlling the Pointer Bounds Checker. See Pointer Bounds Checker builtins, for more information.

-fchkp-check-incomplete-type

Generate pointer bounds checks for variables with incomplete type. Enabled by default.

-fchkp-narrow-bounds

Controls bounds used by Pointer Bounds Checker for pointers to object fields. If narrowing is enabled then field bounds are used. Otherwise object bounds are used. See also -fchkp-narrow-to-innermost-array and -fchkp-first-field-has-own-bounds. Enabled by default.

-fchkp-first-field-has-own-bounds

Forces Pointer Bounds Checker to use narrowed bounds for the address of the first field in the structure. By default a pointer to the first field has the same bounds as a pointer to the whole structure.

-fchkp-flexible-struct-trailing-arrays

Forces Pointer Bounds Checker to treat all trailing arrays in structures as possibly flexible. By default only array fields with zero length or that are marked with attribute bnd_variable_size are treated as flexible.

-fchkp-narrow-to-innermost-array

Forces Pointer Bounds Checker to use bounds of the innermost arrays in case of nested static array access. By default this option is disabled and bounds of the outermost array are used.

-fchkp-optimize

Enables Pointer Bounds Checker optimizations. Enabled by default at optimization levels -O, -O2, -O3.

-fchkp-use-fast-string-functions

Enables use of *_nobnd versions of string functions (not copying bounds) by Pointer Bounds Checker. Disabled by default.

-fchkp-use-nochk-string-functions

Enables use of *_nochk versions of string functions (not checking bounds) by Pointer Bounds Checker. Disabled by default.

-fchkp-use-static-bounds

Allow Pointer Bounds Checker to generate static bounds holding bounds of static variables. Enabled by default.

-fchkp-use-static-const-bounds

Use statically-initialized bounds for constant bounds instead of generating them each time they are required. By default enabled when -fchkp-use-static-bounds is enabled.

-fchkp-treat-zero-dynamic-size-as-infinite

With this option, objects with incomplete type whose dynamically-obtained size is zero are treated as having infinite size instead by Pointer Bounds Checker. This option may be helpful if a program is linked with a library missing size information for some symbols. Disabled by default.

-fchkp-check-read

Instructs Pointer Bounds Checker to generate checks for all read accesses to memory. Enabled by default.

-fchkp-check-write

Instructs Pointer Bounds Checker to generate checks for all write accesses to memory. Enabled by default.

-fchkp-store-bounds

Instructs Pointer Bounds Checker to generate bounds stores for pointer writes. Enabled by default.

-fchkp-instrument-calls

Instructs Pointer Bounds Checker to pass pointer bounds to calls. Enabled by default.

-fchkp-instrument-marked-only

Instructs Pointer Bounds Checker to instrument only functions marked with the bnd_instrument attribute (see Function Attributes). Disabled by default.

-fchkp-use-wrappers

Allows Pointer Bounds Checker to replace calls to built-in functions with calls to wrapper functions. When -fchkp-use-wrappers is used to link a program, the GCC driver automatically links against libmpxwrappers. See also -static-libmpxwrappers. Enabled by default.

-fcf-protection==[full|branch|return|none]

Enable code instrumentation of control-flow transfers to increase program security by checking that target addresses of control-flow transfer instructions (such as indirect function call, function return, indirect jump) are valid. This prevents diverting the flow of control to an unexpected target. This is intended to protect against such threats as Return-oriented Programming (ROP), and similarly call/jmp-oriented programming (COP/JOP).

The value branch tells the compiler to implement checking of validity of control-flow transfer at the point of indirect branch instructions, i.e. call/jmp instructions. The value return implements checking of validity at the point of returning from a function. The value full is an alias for specifying both branch and return. The value none turns off instrumentation.

You can also use the nocf_check attribute to identify which functions and calls should be skipped from instrumentation (see Function Attributes).

Currently the x86 GNU/Linux target provides an implementation based on Intel Control-flow Enforcement Technology (CET). Instrumentation for x86 is controlled by target-specific options -mcet, -mibt and -mshstk (see x86 Options).

-fstack-protector

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

-fstack-protector-all

Like -fstack-protector except that all functions are protected.

-fstack-protector-strong

Like -fstack-protector but includes additional functions to be protected — those that have local array definitions, or have references to local frame addresses.

-fstack-protector-explicit

Like -fstack-protector but only protects those functions which have the stack_protect

attribute.

-fstack-check

Generate code to verify that you do not go beyond the boundary of the stack. You should specify this flag if you are running in an environment with multiple threads, but you only rarely need to specify it in a single-threaded environment since stack overflow is automatically detected on nearly all systems if there is only one stack.

Note that this switch does not actually cause checking to be done; the operating system or the language runtime must do that. The switch causes generation of code to ensure that they see the stack being extended.

You can additionally specify a string parameter: 'no' means no checking, 'generic' means force the use of old-style checking, 'specific' means use the best checking method and is equivalent to bare -fstack-check.

Old-style checking is a generic mechanism that requires no specific target support in the compiler but comes with the following drawbacks:

1. Modified allocation strategy for large objects: they are always allocated dynamically if their size exceeds a fixed threshold. Note this may change the semantics of some code.
2. Fixed limit on the size of the static frame of functions: when it is topped by a particular function, stack checking is not reliable and a warning is issued by the compiler.
3. Inefficiency: because of both the modified allocation strategy and the generic implementation, code performance is hampered.

Note that old-style stack checking is also the fallback method for 'specific' if no target support has been added in the compiler.

'-fstack-check=' is designed for Ada's needs to detect infinite recursion and stack overflows. 'specific' is an excellent choice when compiling Ada code. It is not generally sufficient to protect against stack-clash attacks. To protect against those you want '-fstack-clash-protection'.

-fstack-clash-protection

Generate code to prevent stack clash style attacks. When this option is enabled, the compiler will only allocate one page of stack space at a time and each page is accessed immediately after allocation. Thus, it prevents allocations from jumping over

any stack guard page provided by the operating system.

Most targets do not fully support stack clash protection. However, on those targets -fstack-clash-protection will protect dynamic stack allocations. -fstack-clash-protection may also provide limited protection for static stack allocations if the target supports -fstack-check=specific.

-fstack-limit-register=*reg*

-fstack-limit-symbol=*sym*

-fno-stack-limit

Generate code to ensure that the stack does not grow beyond a certain value, either the value of a register or the address of a symbol. If a larger stack is required, a signal is raised at run time. For most targets, the signal is raised before the stack overruns the boundary, so it is possible to catch the signal without taking special precautions.

For instance, if the stack starts at absolute address '0x80000000' and grows downwards, you can use the flags -fstack-limit-symbol=__stack_limit and -Wl,--defsym,__stack_limit=0x7ffe0000 to enforce a stack limit of 128KB. Note that this may only work with the GNU linker.

You can locally override stack limit checking by using the no_stack_limit function attribute (see Function Attributes).

-fsplit-stack

Generate code to automatically split the stack before it overflows. The resulting program has a discontiguous stack which can only overflow if the program is unable to allocate any more memory. This is most useful when running threaded programs, as it is no longer necessary to calculate a good stack size to use for each thread. This is currently only implemented for the x86 targets running GNU/Linux.

When code compiled with -fsplit-stack calls code compiled without -fsplit-stack, there may not be much stack space available for the latter code to run. If compiling all code, including library code, with -fsplit-stack is not an option, then the linker can fix up these calls so that the code compiled without -fsplit-stack always has a large stack. Support for this is implemented in the gold linker in GNU binutils release 2.21 and later.

-fvtable-verify=[std|preinit|none]

This option is only available when compiling C++ code. It turns on (or off, if using -fvtable-verify=none) the security feature that verifies at run time, for every virtual call, that the vtable pointer through which the call is made is valid for the type of the object,

and has not been corrupted or overwritten. If an invalid vtable pointer is detected at run time, an error is reported and execution of the program is immediately halted.

This option causes run-time data structures to be built at program startup, which are used for verifying the vtable pointers. The options 'std' and 'preinit' control the timing of when these data structures are built. In both cases the data structures are built before execution reaches main. Using -fvtable-verify=std causes the data structures to be built after shared libraries have been loaded and initialized. -fvtable-verify=preinit causes them to be built before shared libraries have been loaded and initialized.

If this option appears multiple times in the command line with different values specified, 'none' takes highest priority over both 'std' and 'preinit'; 'preinit' takes priority over 'std'.

-fvtv-debug

When used in conjunction with -fvtable-verify=std or -fvtable-verify=preinit, causes debug versions of the runtime functions for the vtable verification feature to be called. This flag also causes the compiler to log information about which vtable pointers it finds for each class. This information is written to a file named vtv_set_ptr_data.log in the directory named by the environment variable VTV_LOGS_DIR if that is defined or the current working directory otherwise.

Note: This feature *appends* data to the log file. If you want a fresh log file, be sure to delete any existing one.

-fvtv-counts

This is a debugging flag. When used in conjunction with -fvtable-verify=std or -fvtable-verify=preinit, this causes the compiler to keep track of the total number of virtual calls it encounters and the number of verifications it inserts. It also counts the number of calls to certain run-time library functions that it inserts and logs this information for each compilation unit. The compiler writes this information to a file named vtv_count_data.log in the directory named by the environment variable VTV_LOGS_DIR if that is defined or the current working directory otherwise. It also counts the size of the vtable pointer sets for each class, and writes this information to vtv_class_set_sizes.log in the same directory.

Note: This feature *appends* data to the log files. To get fresh log files, be sure to delete any existing ones.

-finstrument-functions

Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions are called with the address of the current function and its call site. (On some platforms, __builtin_return_address does not work beyond the current function, so the call site information may not be available to the profiling functions otherwise.)

```
void __cyg_profile_func_enter (void *this_fn,
                               void *call_site);
void __cyg_profile_func_exit  (void *this_fn,
                               void *call_site);
```

The first argument is the address of the start of the current function, which may be looked up exactly in the symbol table.

This instrumentation is also done for functions expanded inline in other functions. The profiling calls indicate where, conceptually, the inline function is entered and exited. This means that addressable versions of such functions must be available. If all your uses of a function are expanded inline, this may mean an additional expansion of code size. If you use extern inline in your C code, an addressable version of such functions must be provided. (This is normally the case anyway, but if you get lucky and the optimizer always expands the functions inline, you might have gotten away without providing static copies.)

A function may be given the attribute no_instrument_function, in which case this instrumentation is not done. This can be used, for example, for the profiling functions listed above, high-priority interrupt routines, and any functions from which the profiling functions cannot safely be called (perhaps signal handlers, if the profiling routines generate output or allocate memory).

-finstrument-functions-exclude-file-list=*file*,*file*,...

Set the list of functions that are excluded from instrumentation (see the description of -finstrument-functions). If the file that contains a function definition matches with one of *file*, then that function is not instrumented. The match is done on substrings: if the *file* parameter is a substring of the file name, it is considered to be a match.

For example:

```
-finstrument-functions-exclude-file-list=/bits/stl,include/sys
```

excludes any inline function defined in files whose pathnames contain /bits/stl or include/sys.

If, for some reason, you want to include letter ',' in one of *sym*, write '\,'. For example, -finstrument-functions-exclude-file-list='\,\,tmp' (note the single quote surrounding the option).

-finstrument-functions-exclude-function-list=*sym*,*sym*,...

This is similar to -finstrument-functions-exclude-file-list, but this option sets the list of function names to be excluded from instrumentation. The function name to be matched is its user-visible name, such as vector<int> blah(const vector<int> &), not the internal mangled name (e.g., _Z4blahRSt6vectorIiSaIiEE). The match is done on substrings: if the *sym* parameter is a substring of the function name, it is considered to be a match. For C99 and C++ extended identifiers, the function name must be given in UTF-8, not using universal character names.

-fpatchable-function-entry=*N*[,*M*]

Generate *N* NOPs right at the beginning of each function, with the function entry point before the *M*th NOP. If *M* is omitted, it defaults to 0 so the function entry points to the address just at the first NOP. The NOP instructions reserve extra space which can be used to patch in any desired instrumentation at run time, provided that the code segment is writable. The amount of space is controllable indirectly via the number of NOPs; the NOP instruction used corresponds to the instruction emitted by the internal GCC back-end interface gen_nop. This behavior is target-specific and may also depend on the architecture variant and/or other compilation options.

For run-time identification, the starting addresses of these areas, which correspond to their respective function entries minus *M*, are additionally collected in the __patchable_function_entries section of the resulting binary.

Note that the value of __attribute__ ((patchable_function_entry (N,M))) takes precedence over command-line option -fpatchable-function-entry=N,M. This can be used to increase the area size or to remove it completely on a single function. If N=0, no pad location is recorded.

The NOP instructions are inserted at—and maybe before, depending on *M*—the function entry address, even before the prologue.

Next: Preprocessor Options, Previous: Optimize Options, Up: Invoking GCC [Contents]

[Index]