Intel(R) Thread Checker 3.1 for Linux*

Getting Started Guide

Intel® Thread Checker detects data races, deadlocks, stalls, and other threading issues. It can detect the potential for these errors even if the error does not occur during an analysis session. Use Thread Checker to filter out specific types of diagnostics, identify critical source locations, and get tips to improve the robustness of your parallel software.

Overview

This guide presents a threaded code example and teaches you how to use Intel® Thread Checker to identify and handle threading-related issues. After completing this guide, you should be ready to analyze and repair your own code using Thread Checker.

To quickly start using Thread Checker, print this short guide and walk through the example provided.

Contents

Disclaimer and Legal Information

Revision History

Document Number	Revision Number	Description	Revision Date
313445 US	001	Initial release.	May 2006
313445 US	002	Minor edits.	September 2006

1   *Build the Sample Code*

The primes sample code identifies and counts and records the prime numbers in the range from one to 10,000. Using the POSIX* threads APIs, multiple threads perform the work. However, the threads simultaneously access the same memory location, causing potential data races. As a result, this program may generate incorrect results.

To build the sample code:

1. Copy primes directory to a convenient workspace. By default this samples code directory is installed in: `/opt/intel/itt/tcheck/samples/primes`.

2. Set up the Intel(R) Thread Checker environment

    > source `<path_to_tcheck_bin_directory>/tcheckvars.csh`

    or

    > . `<path_to_tcheck_bin_directory>/tcheckvars.sh`

3. Build the primes executable using one of the following methods:

    To build using the GNU* C/C++ compiler:

    a. Go to your copy of the primes directory.

    b. Enter the following command:
        > make
        This command builds the executable files `primes.gcc` and `primesFixed.gcc` using the switches `-g -o0` which turn on debug information and turn off optimization. These settings enable Thread Checker to perform *binary instrumentation.* Thread Checker instruments the executable to enable monitoring of POSIX* threads API calls and of memory accesses.

    To build using the Intel® C++ Compiler for Linux*:

    a. At the shell command prompt, set up the Intel compiler environment. Enter:
        > source *`<path_to_compiler_bin_directory>`*/iccvars.csh
        or
        > . *`<path_to_compiler_bin_directory>`*/iccvars.sh.

    b. If the Intel(R) Thread Checker was not installed in the standard location, (`/opt/intel/itt`), edit the Makefile so that the variable `ITT_BASE` points to the location of the installed `tcheck` directory.

    c. Go to the `primes` directory and enter the command:
        > make icc
        The compiler builds executable files with extensions `.icc` and `.icc.tc`. The switches `-g -o0` turn on debugging and turn off optimization to enable the most information gathering. The files with extension `.icc.tc` are compiled

with the additional `-tcheck` switch which enables compile-time source instrumentation. This switch enables Thread Checker to provide you with even more information during analysis.

NOTE: The files with extension `.icc` are not compiled with the `-tcheck` switch, so only binary instrumentation is available on those files.

CAUTION: Before running your code in a separate window or after a new login, you must source iccvars.* again.

If you run the primes executable several times, you might see the output such as the following:

```
> ./primes.gcc
Determining primes from 1 - 10000
Found 1228 primes

> ./primes.gcc
Determining primes from 1 - 10000
Found 1229 primes

> ./primes.gcc
Determining primes from 1 - 10000
Found 1229 primes

> ./primes.gcc
Determining primes from 1 - 10000
Found 1227 primes
```

What do you see? Different runs of the same program produce inconsistent results!

The correct number of primes between one and 10,000 is 1,229. In this case, it is relatively easy to see that there is a threading inconsistency. In larger programs, a threading inconsistency can be much more difficult to discern.

Thread Checker can help you locate the threading inconsistency even though it does not appear in every run.

2   *Collect Data*

As in section 1, set up the Intel(R) Thread Checker command line tool environment by sourcing `<path_to_tcheck_bin_directory>/tcheckvars.csh` or `tcheckvars.sh`.

Now, start the command line tool by entering:

```
> tcheck_cl ./primes.gcc
```

You should see the following output:

```
Intel(R) Thread Checker 3.1 command line instrumentation driver
Copyright (c) 2006 Intel Corporation. All rights reserved.
Building project
Instrumenting
 25% primes.gcc       ( All Functions ):..
 75% libc-2.3.2.so    ( Minimal ):....
100% libpthread-0.60.so ( Minimal ):..

Running: <path>/primes/primes.gcc

Determining primes from 1 - 10000
Found 1229 primes

Application finished
....
```

The command line tool begins by performing binary instrumentation of the executable `primes.gcc` and its associated modules. It shows an estimate of the instrumentation completion time along with the modules being instrumented. For large applications, this process can take many minutes. The level of instrumentation is also given.

Since debug information is enabled in `primes.gcc`, Thread Checker uses the highest instrumentation level, **All Functions**, for this file.

Since the pre-compiled libraries do not have debug information, Thread Checker uses only the **Minimal** instrumentation level which only notes calls to the library functions.

Intel(R) Thread Checker creates a directory to store temporary data. By default, the name of the directory is `/tmp/<login_name>_tc_cl_cache`. You can specify an alternate directory with a short or long form of a command line switch by entering:

```
> tcheck_cl -d /home/sample_data primes.gcc
```

 or

```
> tcheck_cl --cache_dir /home/sample_data primes.gcc
```

3 Analyze Results and Correct the Code

After the run, Intel® Thread Checker collects instrumentation data, analyzes it and ties it to any available symbol information. It displays results in a table of diagnostics that should look like this:

ID	Short Description	Severity Name	Count	Context [Best]	Description	1st Access[Best]	2nd Access[Best]
1	Write -> Read data-race	Error	941	primes.c:27	Memory read at primes.c:41 conflicts with a prior memory write at primes.c:42 (flow dependence)	primes.c:42	primes.c:41
2	Write -> Read data-race	Error	941	primes.c:27	Memory read at primes.c:42 conflicts with a prior memory write at primes.c:42 (flow dependence)	primes.c:42	primes.c:42
3	Write -> Write data-race	Error	941	primes.c:27	Memory write at primes.c:42 conflicts with a prior memory write at primes.c:42 (output dependence)	primes.c:42	primes.c:42
4	Write -> Write data-race	Error	1	primes.c:27	Memory write at primes.c:41 conflicts with a prior memory write at primes.c:41 (output dependence)	primes.c:41	primes.c:41
5	Thread termination	Information	1	Whole Program 1	Thread termination at primes.c:61 - includes stack allocation of 10489856 and use of 2332 bytes	primes.c:61	primes.c:61

You are now ready to analyze diagnostics and correct threading inconsistencies in the application.

The first diagnostic in the list is identified by **ID 1**. Here Thread Checker identified a **Write -> Read data-race error**. The **Severity Name** indicates the class of the diagnostic, in this case an **Error**. The **Count** field shows how many times this particular event occurred during the course of the run. The actual count may vary from run to run based on the scheduling of threads.

The **Context[Best]** field presents the context of the diagnostic. The **Best** designation indicates that Thread Checker attempts to display the most complete information. Because debug information was present, the context can be given as the function that begins at **line 27** of the source file **primes.c**.

The **Description** field shows a more complete description of the diagnostic. The data dependence error (flow dependence) was caused by one thread writing a variable in **line 42** of **primes.c**, the **1st Access[Best]**, and then another thread reading that same unprotected variable at **line 41**, the **2nd Access[Best]**.

If you do a similar run using the source instrumented version `primes.icc.tc`, you would see the following first diagnostic:

ID	Short Description	Severity Name	Count	Context [Best]	Description	1st Access[Best]	2nd Access[Best]
1	Write -> Read data -race	Error	737	primes .c:27	Memory read of primeCount at primes.c:41 conflicts with a prior memory write of primeCount at primes.c:42 (flow dependence)	primes .c:42	primes .c:41

Notice two differences in the results. The **Count** field value is different due to different scheduling. The **Description** field names the global variable involved, **primeCount**. This extra information is available due to source instrumentation. With binary instrumentation, the names of global objects are not available. Now look at the source lines involved:

```
38            while ( (number % factor) != 0 ) factor += 2;
39            if ( factor == number )
40            {
41                primes[ primeCount ] = number;
42                primeCount++;
43            }
```

The global variable `primeCount` and the global array `primes[]` are unprotected but are accessed by all four worker threads. By adding a synchronization object to serialize the use of the variables, you can protect shared variables from unpredictable concurrent modifications.

To correct the code and eliminate the inconsistency:

1. Add a global mutex and initialize it in the main program.

2. Make each thread acquire the mutex before reading or writing the variables, as follows:

```
pthread_mutex_t cs;
...

void * findPrimes ( void * arg )
{
    ....

        while ( (number % factor) != 0 ) factor += 2;
        if ( factor == number )
        {
            pthread_mutex_lock(&cs);
            primes[ primeCount ] = number;
            primeCount++;
            pthread_mutex_unlock(&cs);
        }
}
```

The sample code `primesFixed.c` contains the modification. The executable `primesFixed.gcc` is ready for analysis by Thread Checker. If built with the Intel C/C++ Compiler for Linux, `primesFixed.icc` is ready for analysis using binary instrumentation and `primesFixed.icc.tc` is already instrumented by the compiler.

4   *Next Steps*

The following additional resources are available to help you make the most of this version of Intel® Thread Checker:

- **Online Help**. A full listing of available Thread Checker command-line options are available in the help. To access the help, run:
  ```
  > tcheck_cl --help
  ```

- **Diagnostics**. Detailed descriptions of diagnostics, their causes, and possible solutions are provided in `DiagnosticsGuide.pdf` in `tcheck/doc`.

- **Samples.** Additional code examples are available for you to explore. Use them to learn to identify and resolve other types of threading errors.
  Code **Samples** are in the `tcheck/samples` directory.

- **Release Notes.** Key product details, updated information on requirements, technical support, and known limitations is available in the product `Release_Notes.txt` in `tcheck/doc`.

- **Intel® Thread Profiler**. After you check your code with Intel® Thread Checker, use the **Intel® Thread Profiler** to help you improve its performance. Find details about Thread Profiler and other Intel software development products at: http://www.intel.com/software/products/.