Embedded Profiler - User Manual

Contents

1 Introduction

Embedded Profiler is a multiplatform low overhead C/C++ profiler designed to measure performance on embedded system targets. Linux and Windows platforms are supported as well. On Windows, both MinGW (Minimalist GNU for Windows) and MSVC (Microsoft Visual Studio) compilers can be used.

Profiler is implemented as a dynamic library. It is based on automatic instrumentation of functions done by the C/C++ compiler.

Profiling of C/C++ application can be done either automatically or manually. Automatic profiling needs no modification of a source code. Manual profiling requires using the API to specify the parts of the source code to be profiled. The resulting log can be opened in Performance Analyzer, a GUI application on PC with several views designed for comfortable log analysis.

Besides of that Embedded Profiler has low impact to C/C++ applications, it implements unique feature to estimate duration of functions as if they were not profiled.

Embedded Profiler library exports special timer with processor cycles resolution using the API. The timer resolution can be converted to nanoseconds or microseconds as well. This allows C/C++ applications to use Embedded Profiler library as multi-platform high resolution timer.

2 Profiling

Profiling can be done in two different ways:

Automatically
: This approach does not need any modification of a source code. The whole application is profiled automatically. Even constructors and destructors of static global objects are profiled.

Manually
: This approach requires small modification of a source code. The EProfiler API is used to configure embedded profiler and to specify the part of the source code to be profiled.

Embedded profiler supports two different profiling modes:

Call Tree Mode
: This mode generates complete function call tree of the profiled application with duration of each function. The resulting log can be quite huge and can be opened in Performance Analyzer using several views. This mode has very low overhead which can be measured by special command-line tool (see Profiling overhead correction). Using this measured overhead, the Performance Analyzer implements unique feature to estimate duration of functions as if they were not profiled.

Function List Mode
: This mode generates only list of profiled functions with number of callings and total duration. The resulting log is very small and can be opened in Performance Analyzer using one special view.

Because the profiling is based on instrumentation of functions, it is necessary to compile the profiled application before profiling. The next chapters describe how to do it.

2.1 Project preparation

These steps describe how to compile an application to be prepared for profiling using embedded profiler.

Note
You should profile the release version of your application (with all compiler optimization flags etc.), no built-in debug info is needed.

2.1.1 GCC and MinGW compilers

1. For all profiled application modules add following compiler switch:

   ```
   -finstrument-functions
   ```

2. Unnecessary functions can be filtered using:

   ```
   -finstrument-functions-exclude-function-list
   -finstrument-functions-exclude-file-list
   ```

3. For all profiled application modules add the following linker argument:

   ```
   -Wl,-Map=ProjectName.map
   ```

4. Link profiled application with appropriate EProfiler dynamic library:

   - `EProfiler/linux32-gcc-intel/lib/libEProfiler.so` for Linux 32,
   - `EProfiler/linux64-gcc-intel/lib/libEProfiler.so` for Linux 64,
   - `EProfiler/windows32-mingw-intel/lib/libEProfiler.dll.a` for MinGW 32 and
   - `EProfiler/android-gcc-arm/lib/libEProfiler.so` for Android ARM.

Important
Android ARM On Android ARM targets it is required to use kernel module which enables cycle counting.
Make sure that kernel module `EProfiler/android-gcc-arm/module/eprofiler_arm_counter.ko` is properly loaded in memory (e.g. use insmod).
Android currently does not support RPATH and by default loads system libraries only. Make sure that the environment variable `LD_LIBRARY_PATH` contains full path to `EProfiler/android-gcc-arm/lib/`.

2.1.2 MSVC compiler (Windows)

1. For all profiled application modules add the following compiler switches:

   ```
   /GH /Gh
   ```

2. For all profiled application modules add the following linker argument:

   ```
   /MAP:ProjectName.map
   ```

3. Link profiled application against EProfiler dynamic library:

   ```
   EProfiler/windows32-msvc-intel/lib/EProfiler.lib
   ```

2.1.3 MSVC compiler (Windows CE, SH4 processor)

1. For all profiled application modules add the following compiler switch:

   ```
   /callcap
   ```

2. For all profiled application modules add the following linker argument:

   ```
   /MAP:ProjectName.map
   ```

3. Link profiled application against EProfiler dynamic library:

   ```
   EProfiler/windowsce-msvc-sh4/lib/EProfiler.lib
   ```

2.2 Automatic profiling

The following steps describe how to use embedded profiler in automatic profiling:

1. Set environment variable `EPROF_AUTOSTART_ENABLED` (or registry value `HKLM\SOFTWARE\Eccam\EProfiler\AutostartEnabled` for Windows CE build) to `1`.
2. If Function List Mode is required, set environment variable `EPROF_FUNCTION_LIST_ENABLED` (or registry value `HKLM\SOFTWARE\Eccam\EProfiler\FunctionListEnabled` for Windows CE build) to `1`.
3. Optionally set environment variable `EPROF_MEMORY_SIZE` (or registry value `HKLM\SOFTWARE\Eccam\EProfiler\MemorySize` for Windows CE build) to profiler memory size in bytes. Default is `1 MB` for *Call Tree Mode* and cca `40 KB` (memory to store `1024` functions) for *Function List Mode*.
4. Run application.
5. Find out the profiler log in format `eprof_thread_XXXX.epl` for each thread in current directory.

Important
The duration of functions is measured in processor cycles. The measured processor cycles are converted to time using CPU frequency which was set before profiling starts. However the CPU frequency can be changed dynamically on modern processors. Therefore to be sure that converted times are correct, it is necessary to fix processor frequency before profiling. Otherwise the profiling log analysis should be based on processor cycles only.

2.3 Manual profiling

The following steps describe how to use embedded profiler in manual profiling:

1. In application sources, add `#include "EProfiler/include/EProfiler.h"`
2. Optionally call `EProfilerConfigureFile()` or `EProfilerConfigureMemory()` before profiling (per each thread).
3. Call `EProfilerStart()` to start profiling (per each thread).
4. Optionally call `EProfilerFlush()` to write profiler data stored in memory into the file.
5. Call `EProfilerStop()` to stop profiling (per each thread).
6. Make sure that environment variable `EPROF_AUTOSTART_ENABLED` (or registry value `HKLM\SOFTWARE\Eccam\EProfiler\AutostartEnabled` for Windows CE build) is not defined or it is set to `0`.
7. If Function List Mode is required, set environment variable `EPROF_FUNCTION_LIST_ENABLED` (or registry value `HKLM\SOFTWARE\Eccam\EProfiler\FunctionListEnabled` for Windows CE build) to `1`.
8. Run application.
9. Find out the profiler log in format `eprof_thread_XXXX.epl` for each thread in current directory or find out profiler logs according configuration set by `EProfiler::ConfigureProfFile()` call.

Example code:

```
#include "EProfiler/include/EProfiler.h"

...

EProfilerConfigureFile(p_prof_file_name);                    // optional
EProfilerConfigureMemory(prof_memory_size, p_prof_memory);   // optional
```

```
EProfilerStart();

...     // code to profile
EProfilerFlush();    // optional - force flushing of acquired data
...     // code to profile

EProfilerStop();
```

Profiling can be started and stopped at different scope, so it is possible to start profiling in a function and stop anytime after that function has returned. Starting in a function and stopping in any function that is nested in that function is possible as well.

Important
Threads are profiled independently - each has to be explicitly started and stopped and each has its own profiling memory and log file.

2.3.1 EProfiler API

`TEProfilerResult EProfilerConfigureFile(const char* p_prof_file_name)`

Description
Configure profiler file name for the current thread.
This method can be called only before `EProfilerStart()` or after `EProfilerStop()`. If this method is not called, profiler will use default file name.

Parameters
`p_prof_file_name` - Pointer to the profiler file name to use or NULL to use default file name.

Return Value
`EProfilerSuccess` - success
`EProfilerFailure` - failure, the profiler has been already started

`TEProfilerResult EProfilerConfigureMemory(size_t prof_memory_size, void* p_prof_memory)`

Description
Configure profiler memory for the current thread.
This method can be called only before `EProfilerStart()` or after `EProfilerStop()`. If this method is not called, profiler memory will have default size and it will be allocated on heap. Default profiler memory size in call tree mode is 1MB (1048576 bytes). Default profiler memory size in function list mode is 40KB (40960 bytes).

Parameters
`prof_memory_size` - Profiler memory size in bytes to use or 0 to use default memory size.
`p_prof_memory` - Pointer to memory to use or NULL to allocate memory on heap.

Return Value
`EProfilerSuccess` - success
`EProfilerFailure` - failure, the profiler has been already started or given memory size is too small.

`void EProfilerStart()`

Description
Starts profiling for the current thread.

`void EProfilerFlush()`

Description
Flushes profiling memory to the profiler file for the current thread.
This method stores entry and exit of virtual function `InternalFlushing` to measure consumed time by flushing.
It can be called only between `EProfilerStart()` and `EProfilerStop()`.

`void EProfilerStop()`

Description
Stops profiling for the current thread.

3 Performance Analysis

This section describes how to analyse logs with profiling data generated by embedded profiler.

3.1 Resolving symbols in profiled application

The symbols used in profiled application are generated from map files created during project preparation part. The generated symbols are stored in a special symbol file. The symbol file is needed for PC GUI application Performance Analyzer to resolve properly function names.

The following sections describe how to create this symbol file.

3.1.1 GCC and MinGW compilers

Generate symbol file from all map files generated during compilation by command:

- Linux 32

```
PerformanceAnalyzer/linux32-gcc-intel/bin/EProfilerSymGen \
module1.map [moduleN.map...] ProjectName.sym
```

- Linux 64

```
PerformanceAnalyzer/linux64-gcc-intel/bin/EProfilerSymGen \
module1.map [moduleN.map...] ProjectName.sym
```

- MinGW 32

```
PerformanceAnalyzer/windows32-mingw-intel/bin/EProfilerSymGen.exe \
module1.map [moduleN.map...] ProjectName.sym
```

3.1.2 MSVC compiler (Windows)

Generate symbol file from all map files generated during compilation by command:

```
PerformanceAnalyzer/windows32-msvc-intel/bin/EProfilerSymGen.exe \
module1.map [moduleN.map...] ProjectName.sym
```

3.1.3 MSVC compiler (Windows CE)

Generate symbol file from all map files (if `wathunk.map` is present, `.thunk.lis` files are needed too) generated during compilation by command:

```
PerformanceAnalyzer/windows32-msvc-intel/bin/EProfilerSymGen.exe \
module1.map [moduleN.map...] ProjectName.sym
```

Tip
Post-build steps If it is easy enough to set up a post-build step for each project in your building environment it could make generation even easier. Your post-build step just needs to run:

```
./EProfilerSymGen path_to_project_map_files/*.map ProjectName.sym
```

3.2 Performance Analyzer

The Performance Analyzer is PC GUI application for exploring log files generated by embedded profiler. The following steps describe basic usage of this application:

- The Performance Analyzer can be found in directory `PerformanceAnalyzer/platform-name/bin` where *platform-name* can be `linux32-gcc-intel`, `linux64-gcc-intel`, `windows32-msvc-intel` or `windows32-mingw-intel`.
- Run `PerformanceAnalyzer`.
- Select Menu *File* → *Open* to open profiler log with corresponding symbol file.
- There are several view types which can be opened from the menu *View*:

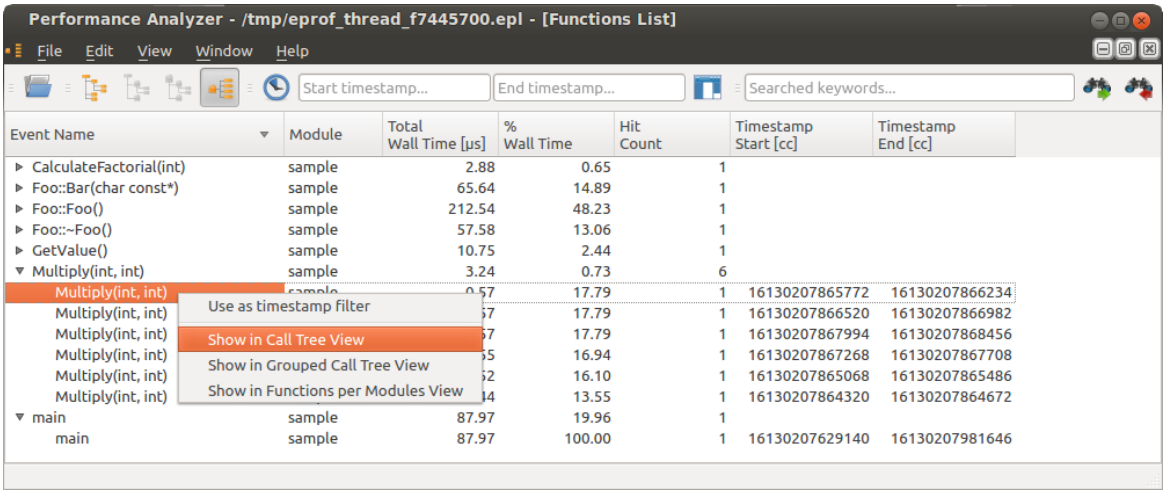
  Call Tree
  Displays function call tree of profiled application. This view is default for Call Tree Mode and it is disabled for Function List Mode.

  Grouped Call Tree
  Similar to Call Tree, but this view groups the same consecutive function calls as a simplification for large logs analysis. This view is disabled for Function List Mode.

  Functions per Modules
  Displays all functions calls grouped by modules and function names. This view is disabled for Function List Mode.

  Functions List
  Displays all functions called by application grouped by function name. This view is default for Function List Mode.

- It is possible to show selected function call in other view using popup menu:



Tip
Suspicious functions can be easily found in the *Function List* view and then shown in their context using the *Call Tree* view.

- Search toolbars allows to search recursively in a tree view by function name.
- Opened log can be filtered using a range defined by start and end timestamp in menu *Edit* → *Set filter*. Filter can be then turned on/off with menu *Edit* → *Timeline Filter*. This function is disabled for Function List Mode.
- The time units can be changed from microseconds to processor cycles by menu item *Edit* → *Set Profiling Parameters* → *Display cycle count [cc] instead of time [µs]*.

- Profiling overhead can be corrected using advanced features.

Note
Android and Windows CE Android and Windows CE platforms do not support Performance Analyzer, use Linux or Windows platforms instead!

4 Additional features

4.1 Filtering logs

The `EProfilerLogConverter` command line tool converts embedded profiler logfile from binary format to text format and vice versa. Therefore if the log contains functions that you are not interested in, it can be easily filtered out using `EProfilerLogConverter` in pipeline with a filter.

For example:

```
./EProfilerLogConverter eprof_log -s ProjectName.sym | \
grep -v 'UnwantedFunction()' | ./EProfilerLogConverter - filtered_eprof_log
```

The filter could be a simple `grep` based on function names or your own filter based on module name, timestamp range, etc.

Filtering can also be useful for reducing size of EProfiler log files.

The `EProfilerLogConverter` can be found in directory `PerformanceAnalyzer/platform-name/bin`.

4.2 Profiling overhead correction

Instrumentation comes with certain overhead which differs on each platform. This overhead is injected into measured duration of functions. As an optional feature, this overhead can be measured with `EProfilerOverhead` command line tool and used in Performance Analyzer to estimate duration of functions as if they were not profiled.

The following steps describe how to measure profiling overhead:

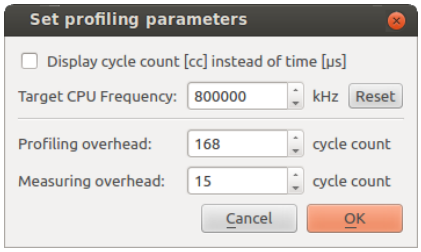
1. Find out the `EProfilerOverhead` in directory `EProfiler/platform-name/bin`.
2. Run application:

   ```
   ./EProfilerOverhead num_of_repetitions num_of_overhead_measurements \
   [-l eprofiler_log_file]
   ```

3. See help for overhead measuring guide:

   ```
   ./EProfilerOverhead --help
   ```

4. Use `profiling overhead` and `measuring overhead` in Performance Analyzer *Edit* → *Set Profiling Parameters* dialog:



4.3 Using EProfiler timer

EProfiler timer interface can be used as a cpu cycle accurate stopwatch. The following steps describe the usage:

1. In application sources, add:

```
#include "EProfiler/include/EProfilerTimer.h"
```

2. Measure elapsed time in processor cycles according to the following example:

```
EProfilerTimer my_timer;
my_timer.Start();
...     // code to measure
const uint64_t duration_in_processor_cycles = my_timer.Stop();
```

3. Link application against appropriate EProfiler dynamic library.

A Step by step tutorial

Example source code:

```
#include <cstdio>
#include <ctime>

class Foo
{
public:
    Foo();
    ~Foo();

    void Bar(const char* p_message);
};

Foo::Foo()
{
    printf("Foo initialization.\n");
}

Foo::~Foo()
{
    printf("Foo deinitilization.\n");
}

void Foo::Bar(const char* p_message)
{
    printf("%s\n", p_message);
```

```
}

int GetValue()
{
    return time(NULL) % 15;
}

int Multiply(int value1, int value2)
{
    int result = 0;
    for (int i = 0; i < value2; ++i)
        result += value1;

    return result;
}

int CalculateFactorial(int value)
{
    int result = 1;
    for (int i = 1; i <= value; ++i)
    {
        result = Multiply(result, i);
    }

    return result;
}

int main()
{
    Foo foo;
    foo.Bar("FooBar");
    const int value = GetValue();
    const int factorial = CalculateFactorial(value);
    printf("Factorial of %d is %d.\n", value, factorial);

    return 0;
}
```

Linux/GCC Compile the example with the following command (assuming `libEProfiler.so` in the current directory):

```
g++ -finstrument-functions -o sample -Wl,-Map=sample.map sample.cpp \
libEProfiler.so
```

Note
Linux 64 When linking 32bit version of EProfiler library, `-m32` flag has to be used:

```
g++ -m32 -finstrument-functions -o sample -Wl,-Map=sample.map sample.cpp \
libEProfiler.so
```

Generate the symbol file (assuming that `EProfilerSymGen` is located in current directory):

```
./EProfilerSymGen sample.map sample.sym
```

For Call Tree Mode, run the example with automatic profiling (assuming `libEProfiler.so` in the current directory):

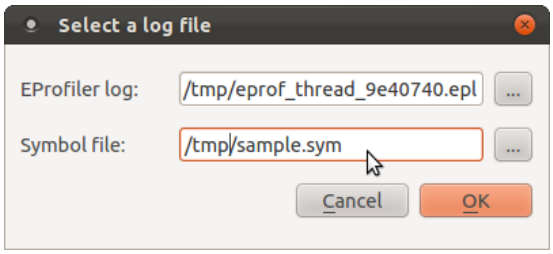
```
LD_LIBRARY_PATH=. EPROF_AUTOSTART_ENABLED=1 ./sample
```

For Function List Mode, run the example with automatic profiling (assuming `libEProfiler.so` in the current directory):

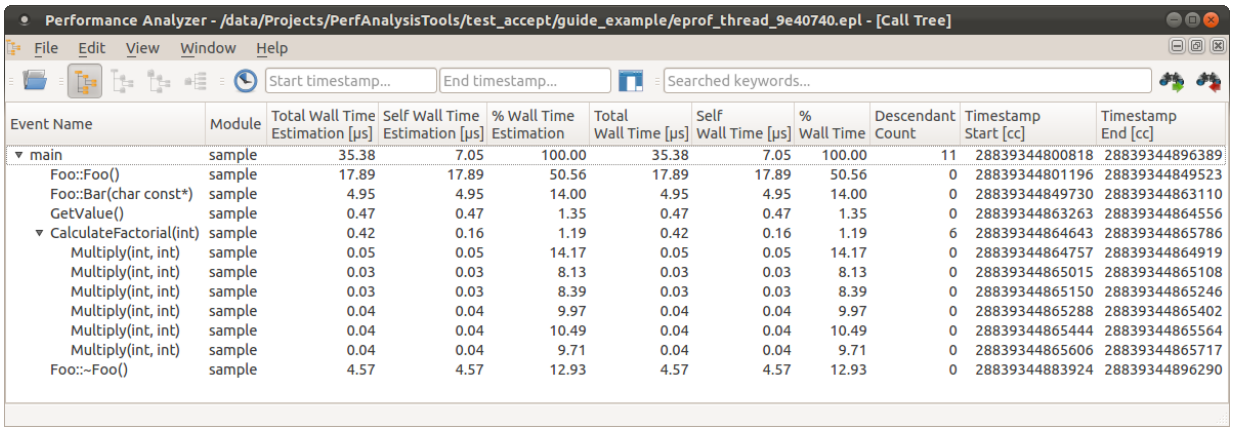
```
LD_LIBRARY_PATH=. EPROF_AUTOSTART_ENABLED=1 EPROF_FUNCTION_LIST_ENABLED=1 ./sample
```

Note
Android ARM Make sure that Android target prerequisites are accomplished.

The profiler log will be saved in the current directory with name `eprof_thread_id.epl`,
e.g. `eprof_thread_9e40740.epl`. This profiler log can be analyzed by Performance Analyzer:

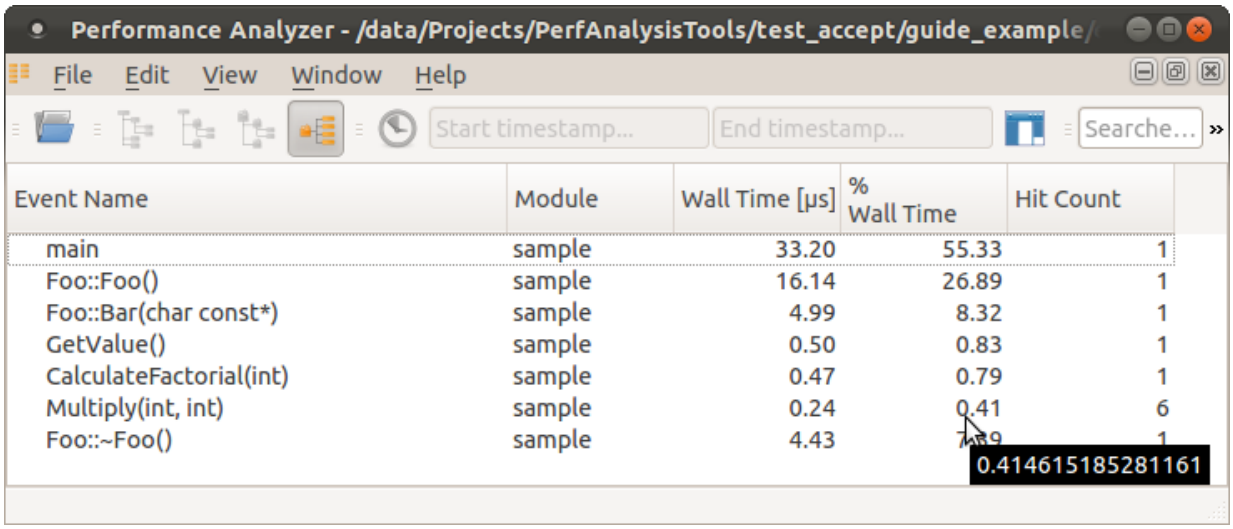


Call Tree Mode:



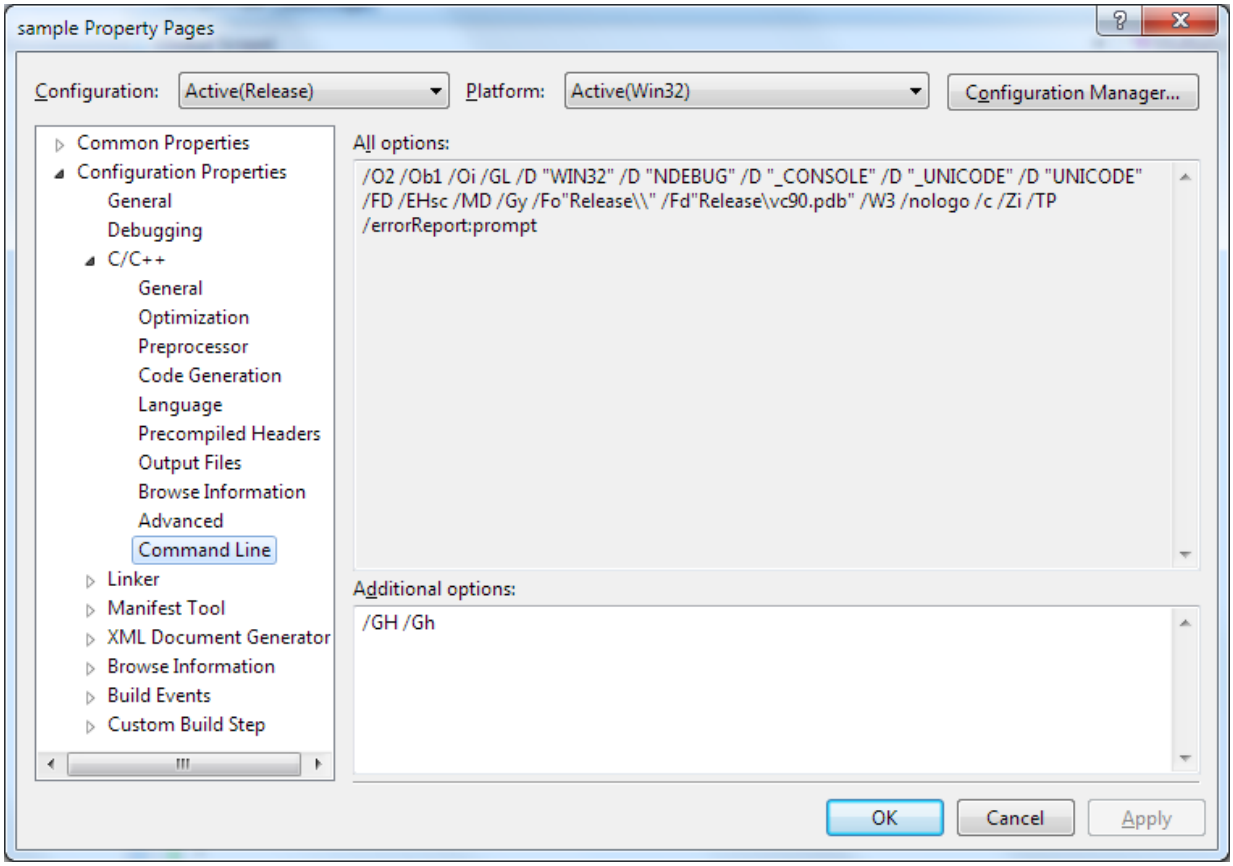
Event Name	Module	Total Wall Time Estimation [µs]	Self Wall Time Estimation [µs]	% Wall Time Estimation	Total Wall Time [µs]	Self Wall Time [µs]	% Wall Time	Descendant Count	Timestamp Start [cc]	Timestamp End [cc]
main	sample	35.38	7.05	100.00	35.38	7.05	100.00	11	28839344800818	28839344896389
Foo::Foo()	sample	17.89	17.89	50.56	17.89	17.89	50.56	0	28839344801196	28839344849523
Foo::Bar(char const*)	sample	4.95	4.95	14.00	4.95	4.95	14.00	0	28839344849730	28839344863110
GetValue()	sample	0.47	0.47	1.35	0.47	0.47	1.35	0	28839344863263	28839344864556
CalculateFactorial(int)	sample	0.42	0.16	1.19	0.42	0.16	1.19	6	28839344864643	28839344865786
Multiply(int, int)	sample	0.05	0.05	14.17	0.05	0.05	14.17	0	28839344864757	28839344864919
Multiply(int, int)	sample	0.03	0.03	8.13	0.03	0.03	8.13	0	28839344865015	28839344865108
Multiply(int, int)	sample	0.03	0.03	8.39	0.03	0.03	8.39	0	28839344865150	28839344865246
Multiply(int, int)	sample	0.04	0.04	9.97	0.04	0.04	9.97	0	28839344865288	28839344865402
Multiply(int, int)	sample	0.04	0.04	10.49	0.04	0.04	10.49	0	28839344865444	28839344865564
Multiply(int, int)	sample	0.04	0.04	9.71	0.04	0.04	9.71	0	28839344865606	28839344865717
Foo::~Foo()	sample	4.57	4.57	12.93	4.57	4.57	12.93	0	28839344883924	28839344896290

Function List Mode:

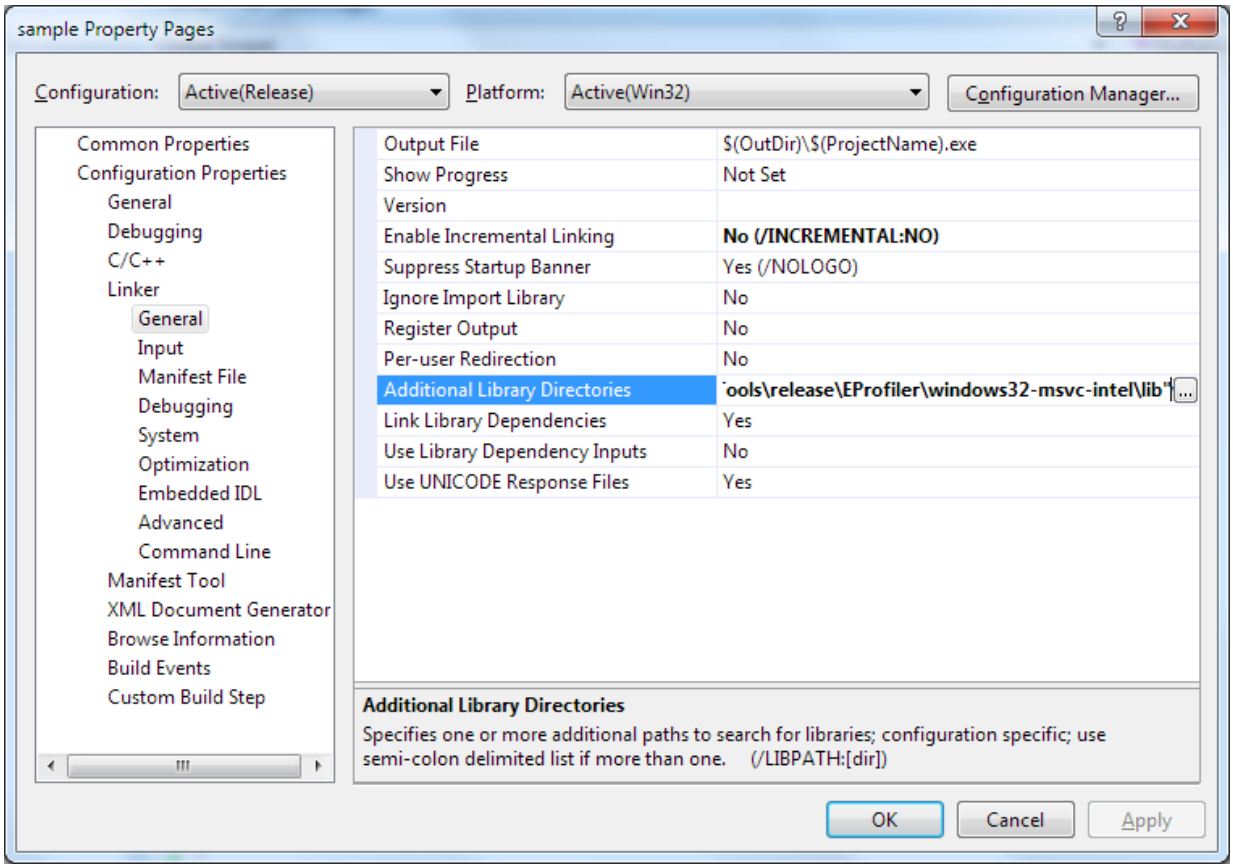


Event Name	Module	Wall Time [µs]	% Wall Time	Hit Count
main	sample	33.20	55.33	1
Foo::Foo()	sample	16.14	26.89	1
Foo::Bar(char const*)	sample	4.99	8.32	1
GetValue()	sample	0.50	0.83	1
CalculateFactorial(int)	sample	0.47	0.79	1
Multiply(int, int)	sample	0.24	0.41	6
Foo::~Foo()	sample	4.43	7.39	1

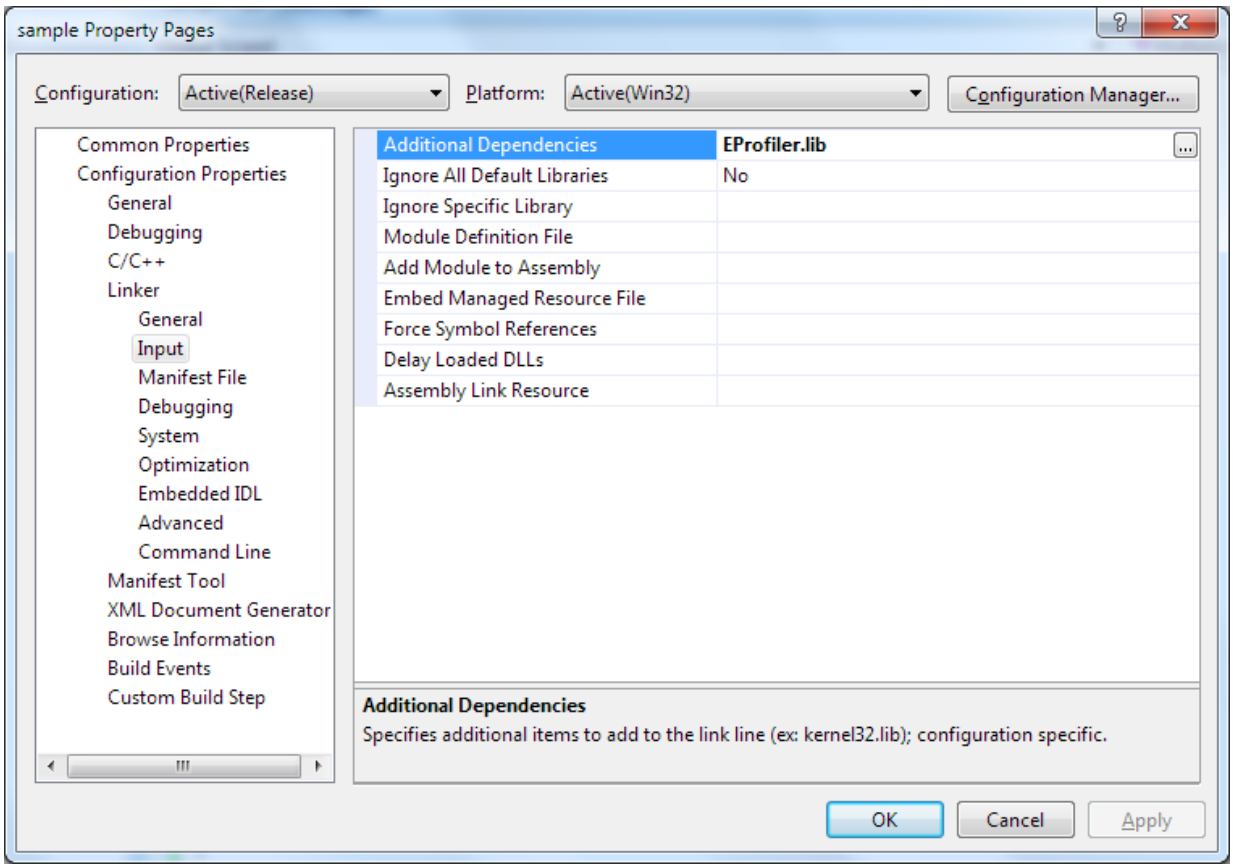
Windows/MSVC Ensure that compiler uses `/Gh /GH` switches:



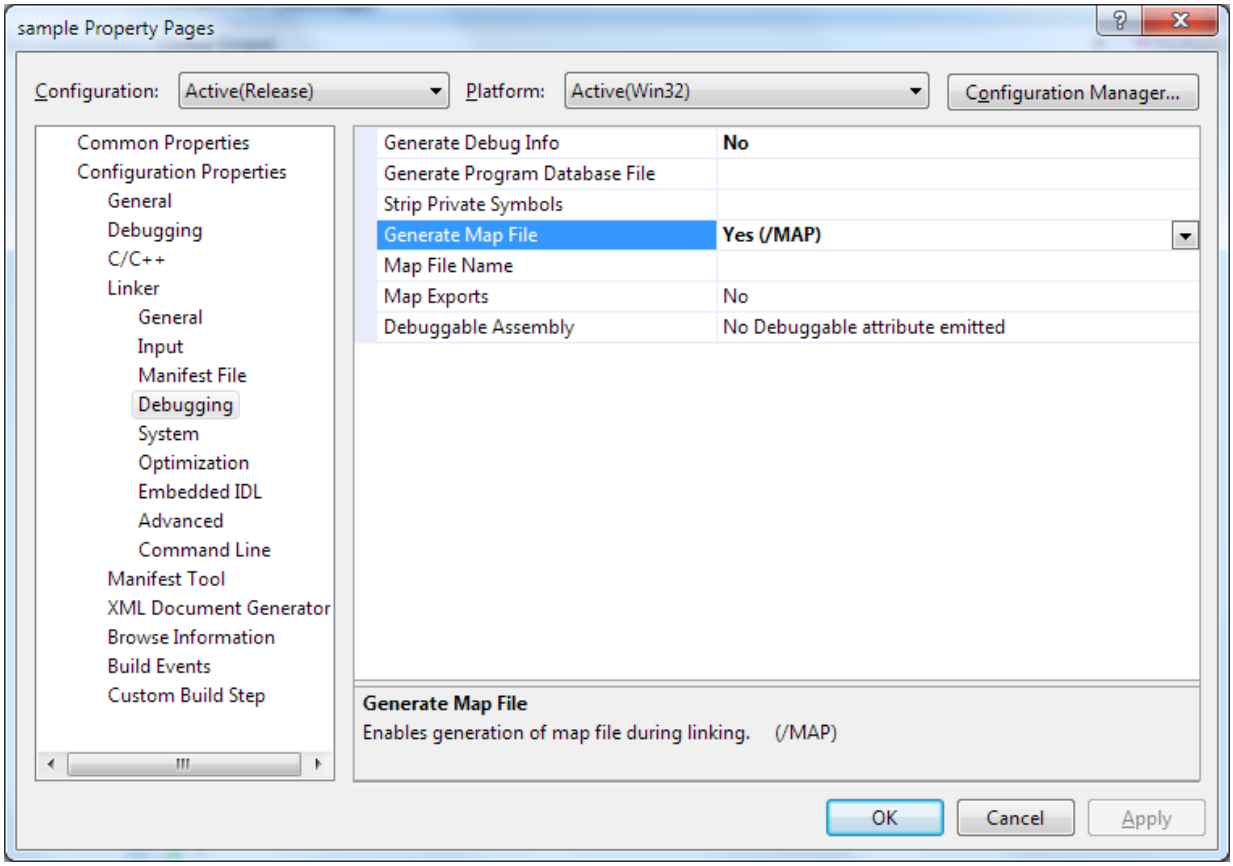
Ensure that linker has access to EProfiler.lib (in Additional Library Directories),



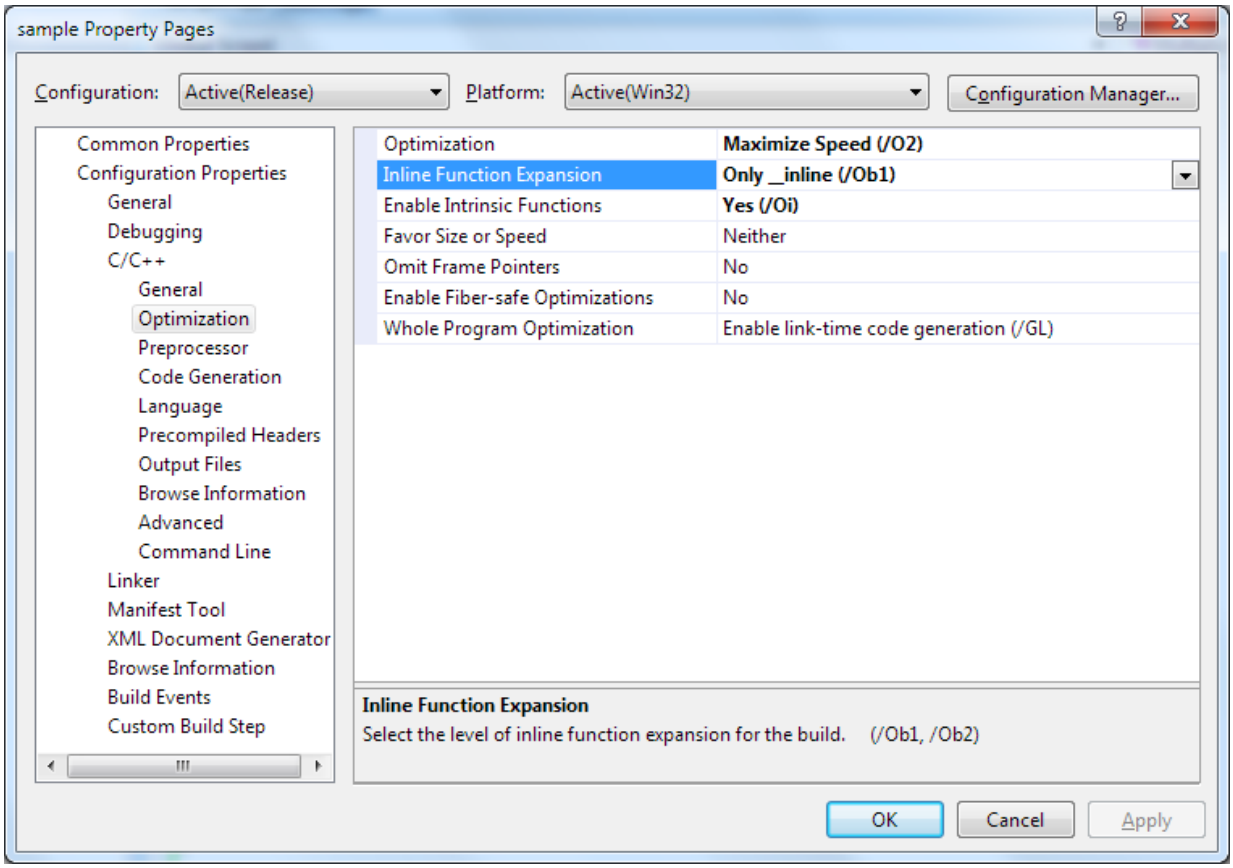
that it links to *EProfiler.lib*



and that it produces the map file:



This sample is so simple that if you use release mode all functions will be inlined and no call tree will be seen in Performance Analyzer. In this case, don't forget to switch off inlining to see all functions in sample:



Generate the symbol file (assuming that `EProfilerSymGen.exe` is in current directory):

```
EProfilerSymGen.exe sample.map sample.sym
```

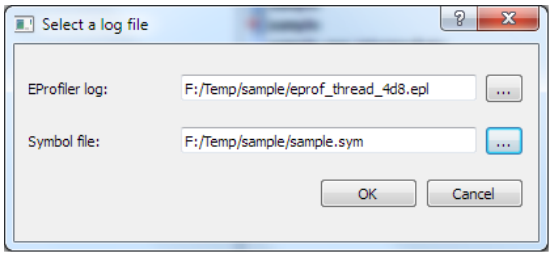
For Call Tree Mode, run the example with automatic profiling (assuming that embedded profiler is installed in `F:\Projects\PerfAnalysisTools\release` directory):

```
set EPROF_AUTOSTART_ENABLED=1
set PATH=%PATH%;^
F:\Projects\PerfAnalysisTools\release\EProfiler\windows32-msvc-intel\bin
sample.exe
```

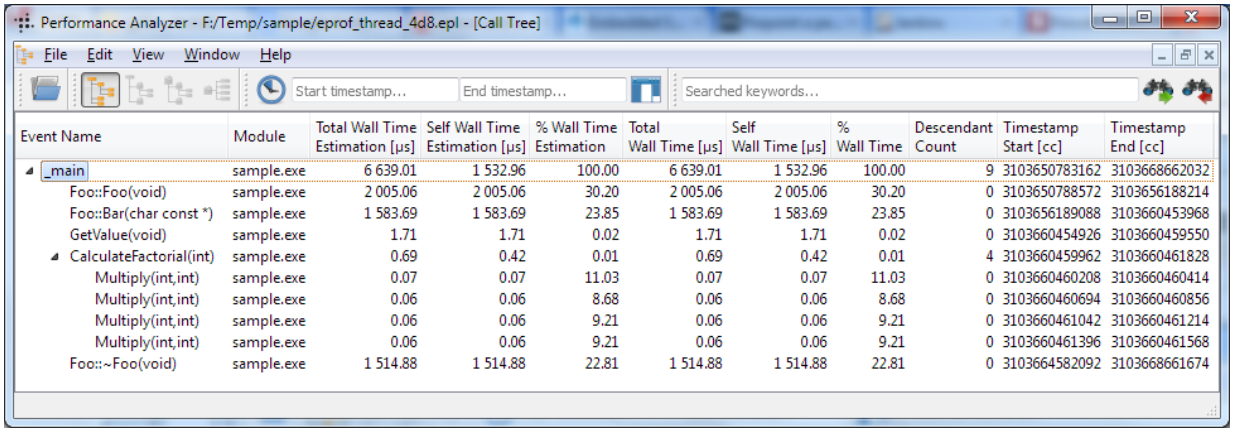
For Function List Mode, run the example with automatic profiling (assuming that embedded profiler is installed in `F:\Projects\PerfAnalysisTools\release` directory):

```
set EPROF_AUTOSTART_ENABLED=1
set EPROF_FUNCTION_LIST_ENABLED=1
set PATH=%PATH%;^
F:\Projects\PerfAnalysisTools\release\EProfiler\windows32-msvc-intel\bin
sample.exe
```

The profiler log will be saved in the current directory with name `eprof_thread_id.epl`, e.g. `eprof_thread_4d8.epl`. This profiler log can be analyzed by Performance Analyzer:

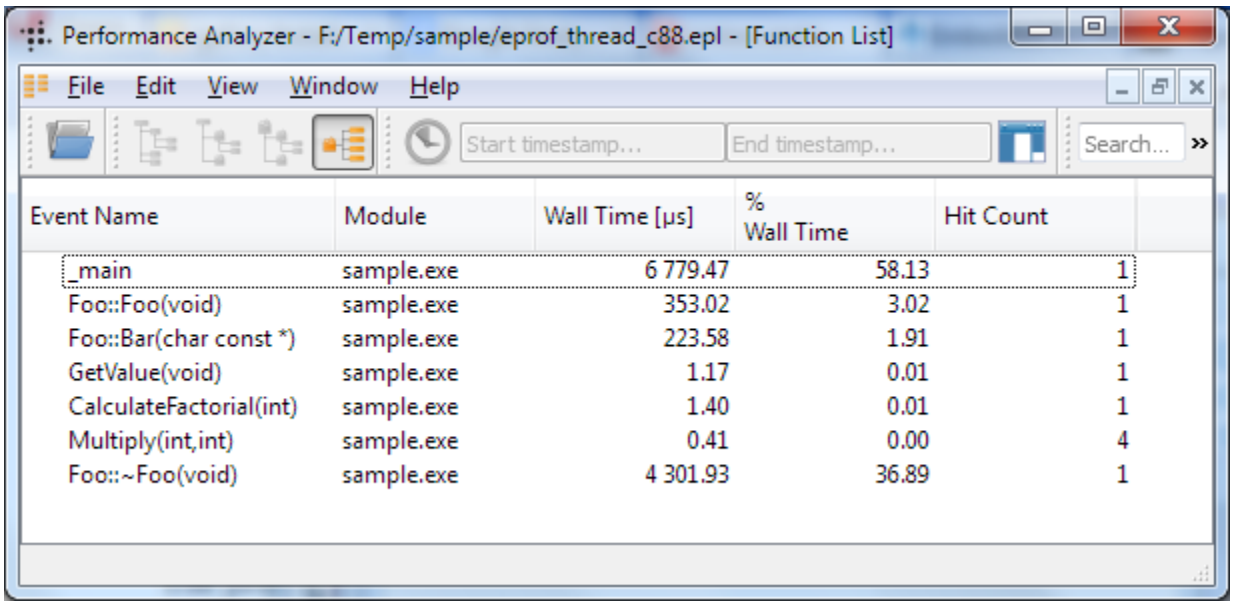


Call Tree Mode:



Event Name	Module	Total Wall Time Estimation [µs]	Self Wall Time Estimation [µs]	% Wall Time Estimation	Total Wall Time [µs]	Self Wall Time [µs]	% Wall Time	Descendant Count	Timestamp Start [cc]	Timestamp End [cc]
_main	sample.exe	6 639.01	1 532.96	100.00	6 639.01	1 532.96	100.00	9	3103650783162	3103668662032
Foo::Foo(void)	sample.exe	2 005.06	2 005.06	30.20	2 005.06	2 005.06	30.20	0	3103650788572	3103656188214
Foo::Bar(char const *)	sample.exe	1 583.69	1 583.69	23.85	1 583.69	1 583.69	23.85	0	3103656189088	3103660453968
GetValue(void)	sample.exe	1.71	1.71	0.02	1.71	1.71	0.02	0	3103660454926	3103660459550
CalculateFactorial(int)	sample.exe	0.69	0.42	0.01	0.69	0.42	0.01	4	3103660459962	3103660461828
Multiply(int,int)	sample.exe	0.07	0.07	11.03	0.07	0.07	11.03	0	3103660460208	3103660460414
Multiply(int,int)	sample.exe	0.06	0.06	8.68	0.06	0.06	8.68	0	3103660460694	3103660460856
Multiply(int,int)	sample.exe	0.06	0.06	9.21	0.06	0.06	9.21	0	3103660461042	3103660461214
Multiply(int,int)	sample.exe	0.06	0.06	9.21	0.06	0.06	9.21	0	3103660461396	3103660461568
Foo::~Foo(void)	sample.exe	1 514.88	1 514.88	22.81	1 514.88	1 514.88	22.81	0	3103664582092	3103668661674

Function List Mode:



Event Name	Module	Wall Time [µs]	% Wall Time	Hit Count
_main	sample.exe	6 779.47	58.13	1
Foo::Foo(void)	sample.exe	353.02	3.02	1
Foo::Bar(char const *)	sample.exe	223.58	1.91	1
GetValue(void)	sample.exe	1.17	0.01	1
CalculateFactorial(int)	sample.exe	1.40	0.01	1
Multiply(int,int)	sample.exe	0.41	0.00	4
Foo::~Foo(void)	sample.exe	4 301.93	36.89	1

Copyright © 2012, Eccam, s.r.o. All rights reserved.