Center for Information Services &
High Performance Computing

VampirTrace 5.12

User Manual

TU Dresden
Center for Information Services and
High Performance Computing (ZIH)
01062 Dresden
Germany

http://www.tu-dresden.de/zih
http://www.tu-dresden.de/zih/vampirtrace

Contact: vampirsupport@zih.tu-dresden.de

Contents

This documentation describes how to apply VampirTrace to an application in order to generate trace files at execution time. This step is called *instrumentation*. It furthermore explains how to control the runtime measurement system during execution (*tracing*). This also includes performance counter sampling as well as selective filtering and grouping of functions.

1. Introduction

VampirTrace consists of a tool set and a runtime library for instrumentation and tracing of software applications. It is particularly tailored to parallel and distributed High Performance Computing (HPC) applications.

The instrumentation part modifies a given application in order to inject additional measurement calls during runtime. The tracing part provides the actual measurement functionality used by the instrumentation calls. By this means, a variety of detailed performance properties can be collected and recorded during runtime. This includes function enter and leave events, MPI communication, OpenMP events, and performance counters.

After a successful tracing run, VampirTrace writes all collected data to a trace file in the Open Trace Format (OTF)[1]. As a result, the information is available for post-mortem analysis and visualization by various tools. Most notably, VampirTrace provides the input data for the Vampir analysis and visualization tool[2].

VampirTrace is included in Open MPI 1.3 and later versions. If not disabled explicitly, VampirTrace is built automatically when installing Open MPI[3].

Trace files can quickly become very large, especially with automatic instrumentation. Tracing applications for only a few seconds can result in trace files of several hundred megabytes. To protect users from creating trace files of several gigabytes, the default behavior of VampirTrace limits the internal buffer to 32 MB per process. Thus, even for larger scale runs the total trace file size will be moderate. Please read Section 3.3 on how to remove or change this limit.

VampirTrace supports various Unix and Linux platforms that are common in HPC nowadays. It is available as open source software under a BSD License.

The following list shows a summary of all instrumentation and tracing features that VampirTrace offers. Note that not all features are supported on all platforms.

[1] http://www.tu-dresden.de/zih/otf
[2] http://www.vampir.eu
[3] http://www.open-mpi.org/faq/?category=vampirtrace

Tracing of user functions ⇒ Chapter 2

- Record function enter and leave events
- Record name and source code location (file name, line)
- Various kinds of instrumentation ⇒ Section 2.2
  - Automatic with many compilers ⇒ Section 2.3
  - Manual using VampirTrace API ⇒ Section 2.4
  - Automatic with tau_instrumentor ⇒ Section 2.5
  - Automatic with Dyninst ⇒ Section 2.6

MPI Tracing ⇒ Chapter 2

- Record MPI functions
- Record MPI communication: participating processes, transferred bytes, tag, communicator

OpenMP Tracing ⇒ Chapter 2

- OpenMP directives, synchronization, thread idle time
- Also hybrid (MPI and OpenMP) applications are supported

Pthread Tracing

- Trace POSIX thread API calls ⇒ Section 4.6
- Also hybrid (MPI and POSIX threads) applications are supported

Java Tracing ⇒ Section 2.8

- Record method calls
- Using JVMTI as interface between VampirTrace and Java Applications

3rd-Party Library tracing ⇒ Section 2.9

- Trace calls to arbitrary third party libraries
- Generate wrapper for library functions based on library's header file(s)
- No recompilation of application or library is required

MPI Correctness Checking ⇒ Section 4.10

- Record MPI usage errors
- Using UniMCI as interface between VampirTrace and a MPI correctness checking tool (e.g. Marmot)

User API

- Manual instrumentation of source code regions ⇒ Section 2.4
- Measurement controls ⇒ Section 2.4.2
- User-defined counters ⇒ Section 4.11
- User-defined marker ⇒ Section 4.12
- User-defined communication ⇒ Section 4.13

Performance Counters ⇒ Sections 4.1 and 4.2

- Hardware performance counters using PAPI, CPC, or NEC SX performance counter
- Resource usage counters using getrusage

Memory Tracing ⇒ Section 4.3

- Trace GLIBC memory allocation and free functions
- Record size of currently allocated memory as counter

I/O Tracing ⇒ Section 4.8

- Trace LIBC I/O calls
- Record I/O events: file name, transferred bytes

CPU ID Tracing ⇒ Section 4.4

- Trace core ID of a CPU on which the calling thread is running
- Record core ID as counter

Fork/System/Exec Tracing ⇒ Section 4.9

- Trace applications calling LIBC's fork, system, or one of the exec functions
- Add forked processes to the trace

Filtering & Grouping ⇒ Chapter 5

- Runtime and post-mortem filter (i.e. exclude functions from being recorded in the trace)
- Runtime grouping (i.e. assign functions to groups for improved analysis)

OTF Output ⇒ Chapter 3

- Writes compressed OTF files
- Output as trace file, statistical summary (profile), or both

2. Instrumentation

To perform measurements with VampirTrace, the user's application program needs to be instrumented, i.e., at specific points of interest (called "events") VampirTrace measurement calls have to be activated. As an example, common events are, amongst others, entering and leaving of functions as well as sending and receiving of MPI messages.

VampirTrace handles this automatically by default. In order to enable the instrumentation of function calls, the user only needs to replace the compiler and linker commands with VampirTrace's wrappers, see Section 2.1 below. VampirTrace supports different ways of instrumentation as described in Section 2.2.

2.1. Compiler Wrappers

All the necessary instrumentation of user functions, MPI, and OpenMP events is handled by VampirTrace's compiler wrappers (vtcc, vtcxx, vtf77, and vtf90). In the script used to build the application (e.g. a makefile), all compile and link commands should be replaced by the VampirTrace compiler wrapper. The wrappers perform the necessary instrumentation of the program and link the suitable VampirTrace library. Note that the VampirTrace version included in Open MPI 1.3 has additional wrappers (mpicc-vt, mpicxx-vt, mpif77-vt, and mpif90-vt) which are like the ordinary MPI compiler wrappers (mpicc, mpicxx, mpif77, and mpif90) with the extension of automatic instrumentation.

The following list shows some examples specific to the parallelization type of the program:

- **Serial programs**: Compiling serial codes is the default behavior of the wrappers. Simply replace the compiler by VampirTrace's wrapper:

original:	`gfortran hello.f90 -o hello`
with instrumentation:	**`vtf90`** `hello.f90 -o hello`

  This will instrument user functions (if supported by the compiler) and link the VampirTrace library.

- **MPI parallel programs**: MPI instrumentation is always handled by means of the PMPI interface, which is part of the MPI standard. This requires the compiler wrapper to link with an MPI-aware version of the VampirTrace library. If your MPI implementation uses special MPI compilers (e.g. mpicc, mpxlf90), you will need to tell VampirTrace's wrapper to use this compiler instead of the serial one:

original:	`mpicc hello.c -o hello`
with instrumentation:	**`vtcc -vt:cc mpicc`** `hello.c -o hello`

MPI implementations without own compilers require the user to link the MPI library manually. In this case, simply replace the compiler by VampirTrace's compiler wrapper:

original:	`icc hello.c -o hello -lmpi`
with instrumentation:	**`vtcc`** `hello.c -o hello -lmpi`

If you want to instrument MPI events only (this creates smaller trace files and less overhead) use the option `-vt:inst manual` to disable automatic instrumentation of user functions (see also Section 2.4).

- **Threaded parallel programs**: When VampirTrace detects OpenMP or Pthread flags on the command line, special instrumentation calls are invoked. For OpenMP events OPARI is invoked for automatic source code instrumentation.

original:	`ifort <-openmp|-pthread> hello.f90 -o hello`
with instrumentation:	**`vtf90`** `<-openmp|-pthread> hello.f90 -o hello`

For more information about OPARI read the documentation available in VampirTrace's installation directory at: `share/vampirtrace/doc/opari/Readme.html`

- **Hybrid MPI/Threaded parallel programs**: With a combination of the above mentioned approaches, hybrid applications can be instrumented:

original:	`mpif90 <-openmp|-pthread> hello.F90 -o hello`
with instrumentation:	**`vtf90 -vt:f90 mpif90`** `<-openmp|-pthread> hello.F90 -o hello`

The VampirTrace compiler wrappers automatically try to detect which parallelization method is used by means of the compiler flags (e.g. `-lmpi`, `-openmp` or `-pthread`) and the compiler command (e.g. `mpif90`). If the compiler wrapper failed to detect this correctly, the instrumentation could be incomplete and an unsuitable VampirTrace library would be linked to the binary. In this case, you should tell the compiler wrapper which parallelization method your program uses by using the switches `-vt:mpi`, `-vt:mt`, and `-vt:hyb` for MPI, multithreaded, and hybrid programs, respectively. Note that these switches do not change the underlying compiler or compiler flags. Use the option `-vt:verbose` to see the command line that the compiler wrapper executes. See Section B.1 for a list of all compiler wrapper options.

The default settings of the compiler wrappers can be modified in the files `share/vampirtrace/vtcc-wrapper-data.txt` (and similar for the other languages) in the installation directory of VampirTrace. The settings include compilers, compiler flags, libraries, and instrumentation types. You could for instance modify the default C compiler from `gcc` to `mpicc` by changing the line `compiler=gcc` to `compiler=mpicc`. This may be convenient if you instrument MPI parallel programs only.

2.2. Instrumentation Types

The wrapper option `-vt:inst <insttype>` specifies the instrumentation type to be used. The following values for `<insttype>` are possible:

- `compinst`
  Fully-automatic instrumentation by the compiler (⇒ Section 2.3)
- `manual`
  Manual instrumentation by using VampirTrace's API (⇒ Section 2.4)
  (needs source-code modifications)
- `tauinst`
  Fully-automatic instrumentation by the tau_instrumentator (⇒ Section 2.5)
- `dyninst`
  Binary-instrumentation with Dyninst (⇒ Section 2.6)

To determine which instrumentation type will be used by default and which instrumentation types are available on your system have a look at the entry `inst_avail` in the wrapper's configuration file (e.g. `share/vampirtrace/vtcc-wrapper-data.txt` in the installation directory of VampirTrace for the C compiler wrapper).

See Section B.1 or type `vtcc -vt:help` for other options that can be passed to VampirTrace's compiler wrapper.

2.3. Automatic Instrumentation

Automatic instrumentation is the most convenient method to instrument your program. If available, simply use the compiler wrappers without any parameters, e.g.:

```
% vtf90 hello.f90 -o hello
```

2.3.1. Supported Compilers

VampirTrace supports following compilers for automatic instrumentation:

- GNU (i.e. gcc, g++, gfortran, g95)
- Intel version ≥10.0 (i.e. icc, icpc, ifort)
- PathScale version ≥3.1 (i.e. pathcc, pathCC, pathf90)
- Portland Group (PGI) (i.e. pgcc, pgCC, pgf90, pgf77)
- SUN Fortran 90 (i.e. cc, CC, f90)
- IBM (i.e. xlcc, xlCC, xlf90)
- NEC SX (i.e. sxcc, sxc++, sxf90)
- OpenUH version ≥4.0 (i.e. uhcc, uhCC, uhf90)

2.3.2. Notes for Using the GNU, Intel, or PathScale Compiler

For these compilers the command `nm` is required to get symbol information of the running application executable. For example on Linux systems, this program is a part of the *GNU Binutils*, which is downloadable from http://www.gnu.org/software/binutils.

To get the application executable for `nm` during runtime, VampirTrace uses the `/proc` file system. As `/proc` is not present on all operating systems, automatic symbol information might not be available. In this case, it is necessary to set the environment variable `VT_APPPATH` to the pathname of the application executable to get symbols resolved via `nm`.

Should any problems emerge to get symbol information automatically, then the environment variable `VT_GNU_NMFILE` can be set to a symbol list file, which is created with the command `nm`, like:

```
% nm hello > hello.nm
```

To get the source code line for the application functions use `nm -l` on Linux systems. VampirTrace will include this information into the trace. Note that the output format of `nm` must be written in BSD-style. See the manual page of `nm` to obtain help for dealing with the output format setting.

2.3.3. Notes on Instrumentation of Inline Functions

Compilers behave differently when they automatically instrument inlined functions. The GNU and Intel ≥10.0 compilers instrument all functions by default when they are used with VampirTrace. They therefore switch off inlining completely, disregarding the optimization level chosen. One can prevent these particular functions from being instrumented by appending the following attribute to function declarations, hence making them able to be inlined (this works only for C/C++):

```
__attribute__ ((__no_instrument_function__))
```

The PGI and IBM compilers prefer inlining over instrumentation when compiling with enabled inlining. Thus, one needs to disable inlining to enable the instrumentation of inline functions and vice versa.

The bottom line is that a function cannot be inlined and instrumented at the same time. For more information on how to inline functions read your compiler's manual.

2.3.4. Instrumentation of Loops with OpenUH Compiler

The OpenUH compiler provides the possibility of instrumenting loops in addition to functions. To use this functionality add the compiler flag `-OPT:instr_loop`. In this case loops induce additional events including the type of loop (e.g. for, while, or do) and the source code location.

2.4. Manual Instrumentation

2.4.1. Using the VampirTrace API

The `VT_USER_START`, `VT_USER_END` calls can be used to instrument any user-defined sequence of statements.

```
Fortran:
           #include "vt_user.inc"
           VT_USER_START('name')
           ...
           VT_USER_END('name')

C:
           #include "vt_user.h"
           VT_USER_START("name");
           ...
           VT_USER_END("name");
```

If a block has several exit points (as it is often the case for functions), all exit points have to be instrumented with `VT_USER_END`, too.

For C++ it is simpler as is demonstrated in the following example. Only entry points into a scope need to be marked. The exit points are detected automatically when C++ deletes scope-local variables.

```
C++:
            #include "vt_user.h"
            {
              VT_TRACER("name");
              ...
            }
```

The instrumented sources have to be compiled with `-DVTRACE` for all three languages, otherwise the `VT_*` calls are ignored. Note that Fortran source files instrumented this way have to be preprocessed, too.

In addition, you can combine this particular instrumentation type with all other types. In such a way, all user functions can be instrumented by a compiler while special source code regions (e.g. loops) can be instrumented by VT's API.

Use VT's compiler wrapper (described above) for compiling and linking the instrumented source code, such as:

- combined with automatic compiler instrumentation:

  ```
  % vtcc -DVTRACE hello.c -o hello
  ```

- without compiler instrumentation:

  ```
  % vtcc -vt:inst manual -DVTRACE hello.c -o hello
  ```

Note that you can also use the option `-vt:inst manual` with non-instrumented sources. Binaries created in this manner only contain MPI and OpenMP instrumentation, which might be desirable in some cases.

2.4.2. Measurement Controls

Switching tracing on/off: In addition to instrumenting arbitrary blocks of code, one can use the `VT_ON`/ `VT_OFF` instrumentation calls to start and stop the recording of events. These constructs can be used to stop recording of events for a part of the application and later resume recording. For example, as is demonstrated in the following C/C++ code snippet, one could not collect trace events during the initialization phase of an application and turn on tracing for the computation part.

```
int main() {
  ...
  VT_OFF();
  initialize();
  VT_ON();
  compute();
  ...
}
```

Furthermore the ”on/off” functionality can be used to control the tracing behavior of VampirTrace and allows to trace only parts of interests. Therefore the amount of trace data can be reduced essentially. To check whether if tracing is enabled or not use the call `VT_IS_ON`.

For further information about limitations have a look at the FAQ D.5.

Trace buffer rewind: An alternative to the ”on/off” functionality is the buffer rewind approach. It is useful when the program should decide dynamically *after* a specific code section (i.e. a time step or iteration) if this section *has been* interesting (i.e. anomalous/slow behavior) and should be recorded to the trace file. The key difference to ”on/off” is that you do not need to know a priori if a section should be recorded.

Use the instrumentation call `VT_SET_REWIND_MARK` at the beginning of a (possibly not interesting) code section. Later, you can decide to rewind the trace buffer to the mark with the call `VT_REWIND`. All recorded trace data between the mark and the rewind call will be dropped. Note, that only one mark can be set at a time. The last call to `VT_SET_REWIND_MARK` will be considered when rewinding the trace buffer. This simplified Fortran code example sketches how the rewind approach can be used:

```
do step=1,number_of_time_steps
  VT_SET_REWIND_MARK()
  call compute_time_step(step)
  if(finished_as_expected) VT_REWIND()
end do
```

Refer to FAQ D.5 for limitations associated with this method.

Intermediate buffer flush: In addition to an automated buffer flush when the buffer is filled, it is possible to flush the buffer at any point of the application. This way you can guarantee that after a manual buffer flush there will be a sequence of the program with no automatic buffer flush interrupting. To flush the buffer you can use the call `VT_BUFFER_FLUSH`.

Intermediate time synchronisation: VampirTrace provides several mechanisms for timer synchronization (⇒ Section 3.7). In addition it is also possible to initiate a timer synchronization at any point of the application by calling `VT_TIMESYNC`. Please note that the user has to ensure that all processes are actual at a synchronized point in the program (e.g. at a barrier). To use this call make sure that the enhanced timer synchronization is activated (set the environment variable `VT_ETIMESYNC` ⇒ Section 3.2).

Intermediate counter update: VampirTrace provides the functionality to collect the values of arbitrary hardware counters. Chosen counter values are automatically recorded whenever an event occurs. Sometimes (e.g. within a long-lasting function) it is desirable to get the counter values at an arbitrary point within the program. To record the counter values at any given point you can call `VT_UPDATE_COUNTER`.

Note: For all three languages the instrumented sources have to be compiled with `-DVTRACE`. Otherwise the `VT_*` calls are ignored.
In addition, if the sources contains further VampirTrace API calls and only the calls for measurement controls shall be disabled, then the sources have to be compiled with `-DVTRACE_NO_CONTROL`, too.

2.5. Source Instrumentation Using PDT/TAU

TAU instrumentation combines the advantages of compiler and manual instrumentation and has further advantages. Like compiler instrumentation it works automatically, like on manual instrumentation you have a filtered set of events, this is especially recommended for C++, because STL-constructor calls are suppressed. Unlike with compiler instrumentation you get an optimized binary – this solves the issue described in Section 2.3.3. In the simpliest case you just run the compiler wrappers with `-vt:inst tauinst` option:

```
% vtcc -vt:inst tauinst hello.c -o hello
```

There is a known issue with the TAU instrumentation in the ⇒ FAQ D.9

Requirements for TAU instrumentation: To work with TAU instrumenation you need the Program Database Toolkit. You have to make sure, to have `cparse` and `tau_instrumentor` in your `$PATH`. The PDToolkit can be downloaded from http://www.cs.uoregon.edu/research/pdt/home.php .

Include/Exclude Lists: `tau_instrumentor` provides a mechanism to include and exclude files or functions from instrumenation. The lists are deposed in a single file, that is announced to `tau_instrumentor` via the option `-f <filename>`. This file contains up to four lists which begin with `BEGIN[_FILE]_<INCLUDE|EXCLUDE>_LIST`. The names in between may contain wildcards as "?", "*', and "#", each entry gets a new line. The lists end with `END[_FILE]_<INCLUDE|EXCLUDE>_LIST`. For further information on selective profiling have a look at the TAU documentation [1]. To announce the file through the compiler wrapper use the option `-vt:tau`:

```
% vtcc -vt:inst tauinst hello.c -o hello \
  -vt:tau '-f <filename>'
```

2.6. Binary Instrumentation Using Dyninst

The option `-vt:inst dyninst` is used with the compiler wrapper to instrument the application during runtime (binary instrumentation), by using Dyninst [2]. Recompiling is not necessary for this kind of instrumentation, but relinking:

```
% vtf90 -vt:inst dyninst hello.o -o hello
```

The compiler wrapper dynamically links the library `libvt-dynatt.so` to the application. This library attaches the *mutator*-program `vtdyn` during runtime which invokes the instrumentation by using Dyninst.

To prevent certain functions from being instrumented you can use the runtime function filtering as explained in Section 5.1. All additional overhead, due to instrumentation of these functions, will be removed.

VampirTrace also allows binary instrumentation of functions located in shared libraries. For this to work a colon-separated list of shared library names has to be given in the environment variable `VT_DYN_SHLIBS`:

```
VT_DYN_SHLIBS=libsupport.so:libmath.so
```

2.6.1. Static Binary Instrumentation

In order to avoid the overhead introduced by Dyninst during runtime, the tool `vtdyn` can be used for binary instrumentation before application launch. To accomplish this, the `-o` or `--output` switch can be used to specify the output binary. Note that the application must be linked to the corresponding VampirTrace library.

[1] http://www.cs.uoregon.edu/research/tau/docs/newguide/ch03s03.html#ManualSelectiveProfiling
[2] http://www.dyninst.org

Example To apply binary instrumentation to the executable `a.out` the following command is nescessary:

```
% vtdyn -o dyninst_a.out ./a.out
```

2.7. Runtime Instrumentation Using VTRun

Besides the already described instrumentation at compile-time, VampirTrace also supports runtime instrumention using the `vtrun` command. Prepending the actual call to the application will transparently add instrumentation support and launch the application. This includes support function instrumentation by Dyninst (Section 2.6) as well as MPI communication tracing. In order to enable instrumentation for user functions the user has to specify the `--dyninst` command line switch.

Example In order to add tracing support to an already existing executable, only a small change to the startup command has to be made. Assuming the usual way of calling the application looks like:

```
% mpirun -np 4 ./a.out
```

By putting the call to `vtrun` directly before the actual application call, instrumention support will be enabled at runtime:

```
% mpirun -np 4 vtrun ./a.out
```

For more information about the tool `vtrun` see Section B.6.

2.8. Tracing Java Applications Using JVMTI

In addition to C, C++, and Fortran, VampirTrace is capable of tracing Java applications. This is accomplished by means of the Java Virtual Machine Tool Interface (JVMTI) which is part of JDK versions 5 and later. If VampirTrace was built with Java tracing support, the library `libvt-java.so` can be used as follows to trace any Java program:

```
% java -agentlib:vt-java ...
```

Or more easier, by replacing the usal Java application launcher `java` by the command `vtjava`:

```
% vtjava ...
```

When tracing Java applications, you probably want to filter out dispensable function calls. Please have a look at Sections 5.1 and 5.2 to learn about different ways for excluding parts of the application from tracing.

2.9. Tracing Calls to 3rd-Party Libraries

VampirTrace is also capable to trace calls to third party libraries, which come with at least one C header file even without the library's source code. If VampirTrace was built with support for library tracing (the CTool library is required), the tool `vtlibwrapgen` can be used to generate a wrapper library to intercept each call to the actual library functions. This wrapper library can be linked to the application or used in combination with the `LD_PRELOAD` mechanism provided by Linux. The generation of a wrapper library is done using the `vtlibwrapgen` command and consists of two steps. The first step generates a C source file, providing the wrapped functions of the library header file:

```
% vtlibwrapgen -g SDL -o SDLwrap.c /usr/include/SDL/*.h
```

This generates the source file `SDLwrap.c` that contains wrapper-functions for all library functions found in the header-files located in /usr/include/SDL/ and instructs VampirTrace to assign these functions to the new group `SDL`.

The generated wrapper source file can be edited in order to add manual instrumentation or alter attributes of the library wrapper. A detailed description can be found in the generated source file or in the header file `vt_libwrap.h` which can be found in the include directory of VampirTrace.

To adapt the library instrumentation it is possible to pass a filter file to the generation process. The rules are like these for normal VampirTrace instrumentation (see Section 5.1), where only 0 (exclude functions) and -1 (generally include functions) are allowed.

The second step is to compile the generated source file:

```
% vtlibwrapgen --build --shared -o libSDLwrap SDLwrap.c
```

This builds the shared library `libSDLwrap.so` which can be linked to the application or preloaded by using the environment variable `LD_PRELOAD`:

```
% LD_PRELOAD=$PWD/libSDLwrap.so <executable>
```

For more information about the tool `vtlibwrapgen` see Section B.5.

3. Runtime Measurement

Running a VampirTrace instrumented application should normally result in an OTF trace file in the current working directory where the application was executed. If a problem occurs, set the environment variable `VT_VERBOSE` to `2` before executing the instrumented application in order to see control messages of the VampirTrace runtime system which might help tracking down the problem.

The internal buffer of VampirTrace is limited to 32 MB per process. Use the environment variables `VT_BUFFER_SIZE` and `VT_MAX_FLUSHES` to increase this limit. Section 3.3 contains further information on how to influence trace file size.

3.1. Trace File Name and Location

The default name of the trace file depends on the operating system where the application is run. On Linux, MacOS and Sun Solaris the trace file will be named like the application, e.g. `hello.otf` for the executable `hello`. For other systems, the default name is `a.otf`. Optionally, the trace file name can be defined manually by setting the environment variable `VT_FILE_PREFIX` to the desired name. The suffix `.otf` will be added automatically.

To prevent overwriting of trace files by repetitive program runs, one can enable unique trace file naming by setting `VT_FILE_UNIQUE` to `yes`. In this case, VampirTrace adds a unique number to the file names as soon as a second trace file with the same name is created. A `*.lock` file is used to count up the number of trace files in a directory. Be aware that VampirTrace potentially overwrites an existing trace file if you delete this lock file. The default value of `VT_FILE_UNIQUE` is `no`. You can also set this variable to a number greater than zero, which will be added to the trace file name. This way you can manually control the unique file naming.

The default location of the final trace file is the working directory at application start time. If the trace file shall be stored in another place, use `VT_PFORM_GDIR` as described in Section 3.2 to change the location of the trace file.

3.2. Environment Variables

The following environment variables can be used to control the measurement of a VampirTrace instrumented executable:

Variable	Purpose	Default
	Global Settings	
VT_APPPATH	Path to the application executable. ⇒ Section 2.3.2	–
VT_BUFFER_SIZE	Size of internal event trace buffer. This is the place where event records are stored, before being written to OTF. ⇒ Section 3.3	32M
VT_CLEAN	Remove temporary trace files?	yes
VT_COMPRESSION	Write compressed trace files?	yes
VT_COMPRESSION_BSIZE	Size of the compression buffer in OTF.	OTF default
VT_FILE_PREFIX	Prefix used for trace filenames.	⇒ Sect. 3.1
VT_FILE_UNIQUE	Enable unique trace file naming? Set to yes, no, or a numerical ID. ⇒ Section 3.1	no
VT_MAX_FLUSHES	Maximum number of buffer flushes. ⇒ Section 3.3	1
VT_MAX_THREADS	Maximum number of threads per process that VampirTrace reserves resources for.	65536
VT_OTF_BUFFER_SIZE	Size of internal OTF buffer. This buffer contains OTF-encoded trace data that is written to file at once.	OTF default
VT_PFORM_GDIR	Name of global directory to store final trace file in.	./
VT_PFORM_LDIR	Name of node-local directory which can be used to store temporary trace files.	/tmp/
VT_THREAD_BUFFER_SIZE	Size of internal event trace buffer for threads. If not defined, the size is set to 10% of VT_BUFFER_SIZE. ⇒ Section 3.3	0
VT_UNIFY	Unify local trace files afterwards?	yes
VT_VERBOSE	Level of VampirTrace related information messages: Quiet (0), Critical (1), Information (2)	1
	Optional Features	
VT_CPUIDTRACE	Enable tracing of core ID of a CPU? ⇒ Section 4.4	no
VT_ETIMESYNC	Enable enhanced timer synchronization? ⇒ Section 3.7	no

Variable	Purpose	Default
VT_ETIMESYNC_INTV	Interval between two successive synchronization phases in s.	120
VT_IOLIB_PATHNAME	Provides an alternative library to use for LIBC I/O calls. ⇒ Section 4.8	–
VT_IOTRACE	Enable tracing of application I/O calls? ⇒ Section 4.8	no
VT_LIBCTRACE	Enable tracing of fork/system/exec calls? ⇒ Section 4.9 calls	yes
VT_MEMTRACE	Enable memory allocation counter? ⇒ Section 4.3	no
VT_MODE	Colon-separated list of VampirTrace modes: Tracing (TRACE), Profiling (STAT). ⇒ Section 3.4	TRACE
VT_MPICHECK	Enable MPI correctness checking via UniMCI?	no
VT_MPICHECK_ERREXIT	Force trace write and application exit if an MPI usage error is detected?	no
VT_MPITRACE	Enable tracing of MPI events?	yes
VT_OMPTRACE	Enable tracing of OpenMP events instrumented by OPARI?	yes
VT_PTHREAD_REUSE	Reuse IDs of terminated Pthreads?	yes
VT_STAT_INTV	Length of interval in ms for writing the next profiling record	0
VT_STAT_PROPS	Colon-separated list of event types that shall be recorded in profiling mode: Functions (FUNC), Messages (MSG), Collective Ops. (COLLOP) or all of them (ALL) ⇒ Section 3.4	ALL
VT_SYNC_FLUSH	Enable synchronized buffer flush? ⇒ Section 3.6	no
VT_SYNC_FLUSH_LEVEL	Minimum buffer fill level for synchronized buffer flush in percent.	80
	Counters	
VT_METRICS	Specify counter metrics to be recorded with trace events as a colon/VT_METRICS_SEP-separated list of names. ⇒ Section 4.1	–
VT_METRICS_SEP	Separator string between counter specifications in VT_METRICS.	:

Variable	Purpose	Default
VT_RUSAGE	Colon-separated list of resource usage counters which shall be recorded. ⇒ Section 4.2	–
VT_RUSAGE_INTV	Sample interval for recording resource usage counters in ms.	100
VT_PLUGIN_CNTR_METRICS	Colon-separated list of plugin counter metrics which shall be recorded. ⇒ Section 4.7	–
	Filtering, Grouping	
VT_DYN_SHLIBS	Colon-separated list of shared libraries for Dyninst instrumentation. ⇒ Section 2.6	–
VT_DYN_IGNORE_NODBG	Disable instrumentation of functions which have no debug information?	no
VT_DYN_DETACH	Detach Dyninst mutator-program `vtdyn` from application process?	yes
VT_FILTER_SPEC	Name of function/region filter file. ⇒ Section 5.1	–
VT_GROUPS_SPEC	Name of function grouping file. ⇒ Section 5.3	–
VT_JAVA_FILTER_SPEC	Name of Java specific filter file. ⇒ Section 5.2	–
VT_GROUP_CLASSES	Create a group for each Java class automatically?	yes
VT_ONOFF_CHECK_STACK_BALANCE	Check stack level balance when switching tracing on/off. ⇒ Section 2.4.2	yes
VT_MAX_STACK_DEPTH	Maximum number of stack level to be traced. (0 = unlimited)	0
	Symbol List	
VT_GNU_NM	Command to list symbols from object files. ⇒ Section 2.3	nm
VT_GNU_NMFILE	Name of file with symbol list information. ⇒ Section 2.3	–

The variables VT_PFORM_GDIR, VT_PFORM_LDIR, VT_FILE_PREFIX may contain (sub)strings of the form $XYZ or ${XYZ} where XYZ is the name of another environment variable. Evaluation of the environment variable is done at measurement runtime.

When you use these environment variables, make sure that they have the same value for all processes of your application on **all** nodes of your cluster. Some cluster environments do not automatically transfer your environment when executing parts of your job on remote nodes of the cluster, and you may need to explicitly set and export them in batch job submission scripts.

3.3. Influencing Trace Buffer Size

The default values of the environment variables `VT_BUFFER_SIZE` and `VT_MAX_FLUSHES` limit the internal buffer of VampirTrace to 32 MB per process and the number of times that the buffer is flushed to 1, respectively. Events that are to be recorded after the limit has been reached are no longer written into the trace file. The environment variables apply to every process of a parallel application, meaning that applications with *n* processes will typically create trace files *n* times the size of a serial application.

To remove the limit and get a complete trace of an application, set `VT_MAX_FLUSHES` to `0`. This causes VampirTrace to always write the buffer to disk when it is full. To change the size of the buffer, use the environment variable `VT_BUFFER_SIZE`. The optimal value for this variable depends on the application which is to be traced. Setting a small value will increase the memory available to the application, but will trigger frequent buffer flushes by VampirTrace. These buffer flushes can significantly change the behavior of the application. On the other hand, setting a large value, like `2G`, will minimize buffer flushes by VampirTrace, but decrease the memory available to the application. If not enough memory is available to hold the VampirTrace buffer and the application data, parts of the application may be swapped to disk, leading to a significant change in the behavior of the application.

In multi-threaded applications a single buffer cannot be shared across a process and the associated threads for performance reasons. Thus independent buffers are created for every process and thread, at which the process buffer size is 70% and the thread buffer size is 10% of the value set in `VT_BUFFER_SIZE`. The buffer size of processes and threads can be explicitly specified setting the environment variable `VT_THREAD_BUFFER_SIZE`, which defines the buffer size of a thread, whereas the buffer size of a process is then defined by the value of `VT_BUFFER_SIZE`.

Note that you can decrease the size of trace files significantly by using the runtime function filtering as explained in Section 5.1.

3.4. Profiling an Application

Profiling an application collects aggregated information about certain events during a program run, whereas tracing records information about individual events. Profiling can therefore be used to get a summary of the program activity and to detect events that are called very often. The profiling information can also be used to generate filter rules to reduce the trace file size (⇒ Section 5.1).

To profile an application set the variable `VT_MODE` to `STAT`. Setting `VT_MODE` to `STAT:TRACE` tells VampirTrace to perform tracing and profiling at the same time. By setting the variable `VT_STAT_PROPS` the user can influence whether functions, messages, and/or collective operations shall be profiled. See Section 3.2 for information about these environment variables.

3.5. Unification of Local Traces

After a run of an instrumented application the traces of the single processes need to be *unified* in terms of timestamps and event IDs. In most cases, this happens automatically. If the environment variable `VT_UNIFY` is set to `no` or under certain circumstances it is necessary to perform unification of local traces manually. To do this, use the following command:

```
% vtunify <prefix>
```

If VampirTrace was built with support for OpenMP and/or MPI, it is possible to speedup the unification of local traces significantly. To distribute the unification on multible processes the MPI parallel version `vtunify-mpi` can be used as follow:

```
% mpirun -np <nranks> vtunify-mpi <prefix>
```

Furthermore, both tools `vtunify` and `vtunify-mpi` are capable to open additional OpenMP threads for unification. The number of threads can be specified by the `OMP_NUM_THREADS` environment variable.

3.6. Synchronized Buffer Flush

When tracing an application, VampirTrace temporarily stores the recorded events in a trace buffer. Typically, if a buffer of a process or thread has reached its maximum fill level, the buffer has to be flushed and other processes or threads maybe have to wait for this process or thread. This will result in an asynchronous runtime behavior.
To avoid this problem, VampirTrace provides a buffer flush in a synchronized

manner. That means, if one buffer has reached its minimum buffer fill level VT_SYNC_FLUSH_LEVEL (⇒ Section 3.2), all buffers will be flushed. This buffer flush is only available at appropriate points in the program flow. Currently, VampirTrace makes use of all MPI collective functions associated with MPI_COMM_WORLD. Use the environment variable VT_SYNC_FLUSH to enable synchronized buffer flush.

3.7. Enhanced Timer Synchronization

Especially on cluster environments, where each process has its own local timer, tracing relies on precisely synchronized timers. Therefore, VampirTrace provides several mechanisms for timer synchronization. The default synchronization scheme is a linear synchronization at the very begin and the very end of a trace run with a master-slave communication pattern.
However, this way of synchronization can become to imprecise for long trace runs. Therefore, we recommend the usage of the enhanced timer synchronization scheme of VampirTrace. This scheme inserts additional synchronization phases at appropriate points in the program flow. Currently, VampirTrace makes use of all MPI collective functions associated with MPI_COMM_WORLD.
To enable this synchronization scheme, a LAPACK library with C wrapper support has to be provided for VampirTrace and the environment variable VT_ETIMESYNC (⇒ Section 3.2) has to be set before the tracing.
The length of the interval between two successive synchronization phases can be adjusted with VT_ETIMESYNC_INTV.
The following LAPACK libraries provide a C-LAPACK API that can be used by VampirTrace for the enhanced timer synchronization:

- CLAPACK CLAPACK[1]
- AMD ACML
- IBM ESSL
- Intel MKL
- SUN Performance Library

Note: Systems equipped with a global timer do not need timer synchronization.

Note: It is recommended to combine enhanced timer synchronization and synchronized buffer flush.

[1] www.netlib.org/clapack

Note: Be aware that the asynchronous behavior of the application will be disturbed since VampirTrace makes use of asynchronous MPI collective functions for timer synchronization and synchronized buffer flush.
Only make use of these approaches, if your application does not rely on an asynchronous behavior! Otherwise, keep this fact in mind during the process of performance analysis.

3.8. Environment Configuration Using VTSetup

In order to ease the process of configuring the runtime environment, the graphical tool `vtsetup` has been added to the VampirTrace toolset. With the help of a graphical user interface, required environment variables can be configured. The following option categories can be managed:

- **General Trace Settings**: Configre the name of the executable as well as the trace filename and set the trace buffer size.
- **Optional Trace Features**: Activate optional trace features, e.g. I/O tracing and tracing of memory usage.
- **Counters**: Activate PAPI counter and resource usage counter.
- **Filtering and Grouping**: Guided setup of filters and function group definitions.

Furthermore, the user is granted more fine-grained control by activating the *Advanced View* button. The configuration can be saved to an XML file. After successfull configuration, the application can be launched directly or a script can be generated for manual execution.

4. Recording Additional Events and Counters

4.1. Hardware Performance Counters

If VampirTrace has been built with hardware counter support (⇒ Appendix A), it is capable of recording hardware counter information as part of the event records. To request the measurement of certain counters, the user is required to set the environment variable `VT_METRICS`. The variable should contain a colon-separated list of counter names or a predefined platform-specific group.

The user can leave the environment variable unset to indicate that no counters are requested. If any of the requested counters are not recognized or the full list of counters cannot be recorded due to hardware resource limits, program execution will be aborted with an error message.

PAPI Hardware Performance Counters

If the PAPI library is used to access hardware performance counters, metric names can be any PAPI preset names or PAPI native counter names. For example, set

```
VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM:!CPU_TEMP1
```

to record the number of floating point instructions and level 2 cache misses (PAPI preset counters), cpu temperature from the lm_sensors component. The leading exclamation mark let CPU_TEMP1 be interpreted as absolute value counter. See Section C.1 for a full list of PAPI preset counters.

CPC Hardware Performance Counters

On Sun Solaris operating systems VampirTrace can make use of the CPC performance counter library to query the processor's hardware performance counters. The counters which are actually available on your platform can be queried with the tool `vtcpcavail`. The listed names can then be used within `VT_METRICS` to tell VampirTrace which counters to record.

NEC SX Hardware Performance Counters

On NEC SX machines VampirTrace uses special register calls to query the processor's hardware counters. Use `VT_METRICS` to specify the counters that have to be recorded. See Section C.3 for a full list of NEC SX hardware performance counters.

4.2. Resource Usage Counters

The Unix system call `getrusage` provides information about consumed resources and operating system events of processes such as user/system time, received signals, and context switches.

If VampirTrace has been built with resource usage support, it is able to record this information as performance counters to the trace. You can enable tracing of specific resource counters by setting the environment variable `VT_RUSAGE` to a colon-separated list of counter names, as specified in Section C.4. For example, set

```
VT_RUSAGE=ru_stime:ru_majflt
```

to record the system time consumed by each process and the number of page faults. Alternatively, one can set this variable to the value `all` to enable recording of all 16 resource usage counters. Note that not all counters are supported by all Unix operating systems. Linux 2.6 kernels, for example, support only resource information for six of them. See Section C.4 and the manual page of `getrusage` for details.

The resource usage counters are not recorded at every event. They are only read if 100 ms have passed since the last sampling. The interval can be changed by setting `VT_RUSAGE_INTV` to the number of desired milliseconds. Setting `VT_RUSAGE_INTV` to zero leads to sampling resource usage counters at every event, which may introduce a large runtime overhead. Note that in most cases the operating system does not update the resource usage information at the same high frequency as the hardware performance counters. Setting `VT_RUSAGE_INTV` to a value less than 10 ms does usually not improve the granularity.

Be aware that, when using the resource usage counters for multi-threaded programs, the information displayed is valid for the whole process and not for each single thread.

4.3. Memory Allocation Counter

The GNU LIBC implementation provides a special hook mechanism that allows intercepting all calls to memory allocation and free functions (e.g. `malloc`, realloc, free). This is independent from compilation or source code access, but relies on the underlying system library.

If VampirTrace has been built with memory-tracing support (⇒ Appendix A), VampirTrace is capable of recording memory allocation information as part of the event records. To request the measurement of the application's allocated memory, the user must set the environment variable VT_MEMTRACE to yes.

Note: This approach to get memory allocation information requires changing internal function pointers in a non-thread-safe way, so VampirTrace currently does not support memory tracing for thread-able programs, e.g., programs parallelized with OpenMP or Pthreads!

4.4. CPU ID Counter

The GNU LIBC implementation provides a function to determine the core id of a CPU on which the calling thread is running. VampirTrace uses this functionality to record the current core identifier as counter. This feature can be activated by setting the environment variable VT_CPUIDTRACE to yes.

Note: To use this feature you need the GNU LIBC implementation at least in version 2.6.

4.5. NVIDIA CUDA Runtime API and Kernels

When tracing CUDA applications, only user events and functions are recorded, which are automatically or manually instrumented. CUDA-Runtime API functions will not be traced by default. To enable tracing of CUDA runtime API functions and asynchronous CUDA tasks (like kernel execution and asynchronous memory copies), build VampirTrace with CUDA support and set the environment variable VT_CUDARTTRACE to yes.

Every CUDA stream, which is executed on a cuda-capable device and used during program execution, creates an own thread. "CUDA-Threads" can contain communication and kernel events and have the following notation:

```
CUDA[device] process:thread
```

To ensure measurement of correct data rates for synchronous CUDA memory copies, VampirTrace inserts a CUDA synchronization before. Otherwise the CUDA memory copy call would do the synchronization and it was not possible to get correct transfer rates.

As kernel execution and asynchronous memory copies are not executed directly, they will be buffered until a synchronizing CUDA Runtime API function call or the programs exit. The buffer size can be specified in bytes (default: 8192) with the environment variable `VT_CUDATRACE_BUFFER_SIZE`.

Several new region groups have been introduced:

CUDART_API	CUDA runtime API calls
CUDA_SYNC	CUDA synchronization
CUDA_KERNEL	CUDA kernels/functions can only appear on "CUDA-Threads"
CUDA_IDLE	GPU idle time – the CUDA device does not run any kernel currently (can only appear in one stream of the device)
VT_CUDA	VampirTrace overhead (write CUDA events, check current device, etc.)

Additional feature switches (environment variables) to customize CUDA runtime tracing:

`VT_CUDATRACE_KERNEL` (default: `yes`)
: Tracing of CUDA kernels is enabled/disabled.

`VT_CUDATRACE_MEMCPYASYNC` (default: `yes`)
: Tracing of asynchronous CUDA memory copies is enabled/disabled.

`VT_CUDATRACE_IDLE` (default: `no`)
: Show the GPU idle time on a CUDA stream, if set to `yes`.

`VT_CUDATRACE_GPUMEMUSAGE` (default: `no`)
: Visualize GPU memory usage as counter "gpu_mem_usage", if set to `yes`.

`VT_CUDATRACE_SYNC` (default: `yes` or `3`)
: Controls how VampirTrace handles synchronizing CUDA API calls, especially *cudaMemcpy* and *cudaThreadSynchronize*. At level 0 only the CUDA calls will be executed, messages will be displayed from the beginning to the end of the *cudaMemcpy*, regardless how long the *cudaMemcpy* call has to wait for a kernel until the actual data transfer starts. At level 1 the *cudaMemcpy* will be split into an additional synchronization and the actual data transfer in order to monitor the data transfer correctly. The additional synchronization does not affect the program execution significantly and will not be shown in the trace. At level 2 the additional synchronization will be exposed to the user. This allows a better view on the application execution, showing how much time is actually spent waiting for a kernel to complete during synchronization. Level 3 will further use the synchronization to flush the internal task buffer and perform a timer synchronization between GPU und and host. This introduces a minimal overhead but increases timer precision and prevents flushes elsewhere in the trace.

VT_CUPTI_METRICS (default: "")
: Capture CUDA CUPTI counters. Metrics are separated by default with ":" or user specified by VT_METRICS_SEP.
Example: VT_CUPTI_METRICS=local_store:local_load

VT_CUPTI_SAMPLING (default: no)
: Poll for CUPTI counter values during kernel execution, if set to yes.

VT_CUPTI_API_CALLBACK (default: no)
: Use CUPTI callback API to intercept CUDA runtime calls.

VT_GPUTRACE_ERROR (default: no)
: Print out an error message and exit the program, if a function call to a GPU library does not return succesfully. The default is just a warning message without program exit.

VT_GPUTRACE_DEBUG (default: no)
: Do not cleanup all GPU ressources (profiling events, contexts, event groups), as they might have been already implicitly cleaned up by the GPU runtime.

Until CUDA Runtime Version 4.0 and CUDA Driver for Linux 270.41.19 the usage of CUDA events between asynchronous tasks serializes their on-device execution. This seems to be a bug, which has already been reported to NVIDIA. As VampirTrace uses CUDA events for time measurement and asynchronous tasks may overlap (depends on the CUDA device capability), there might be a sensible impact on the program flow. The current workaround is to disable tracing of kernels and/or asynchronous memory copies via the given environment variables.

CUDA runtime API Counter

If VT_CUDATRACE_GPUMEMUSAGE is enabled, *cudaMalloc* and *cudaFree* functions will be tracked to write the GPU memory usage counter gpu_mem_usage.

There are three counters, which provide some information about the kernel grid, block and thread compostion (blocks_per_grid, threads_per_block, threads_per_kernel).

CUDA Performance Counters – CUPTI Events

To capture performance counters in CUDA applications, CUPTI metrics can be specified with the environment variable VT_CUPTI_METRICS. Metrics are separated by default with ":" or user specified by VT_METRICS_SEP. The *CUPTI User's Guide* provides information about the available counters. Alternatively set VT_CUPTI_METRICS=help to show a list of available counters (help_long to print the counter description as well).

Tracing CUDA runtime API via CUPTI Callbacks

As there are systems, that does not support dynamic libraries, the CUDA runtime API can be traced via the CUPTI callback interface, implemented in VampirTrace.

If tracing via CUPTI callbacks is enabled (`VT_CUPTI_API_CALLBACK=yes`) and the CUDA runtime wrapper has been configured into the VampirTrace libraries, the CUDA runtime library should be preloaded to reduce tracing overhead (`LD_PRELOAD=libcudart.so`).

Currently CUPTI does not support tracing of asynchronous tasks. If tracing of kernels or asynchronous memory copies is enabled, they will be synchronized directly after the call to retrieve their runtime. This may be improved in future releases.

Compile and Link CUDA applications

Use the VampirTrace compiler wrapper `vtnvcc` instead of `nvcc` to compile the CUDA application, which does automatic source code instrumenation.

GCC4.3 and OpenMP:
Use the flags `-vt:opari -nodecl -Xcompiler=-fopenmp` with `vtnvcc` to compile the OpenMP CUDA application.

CUDA 3.1:
The CUDA runtime library 3.1 creates a conflict with `zlib`. A workaround is to replace all `gcc/g++` calls with the VampirTrace compiler wrappers (`vtcc/vtc++`) and pass the following additional flags to `nvcc` for compilation of the kernels:

```
-I$VT_INSTALL_PATH/include/vampirtrace
-L$VT_INSTALL_PATH/lib
-Xcompiler=-g,-finstrument-functions,-pthread
-lvt -lotf -lcudart -lz -ldl -lm
```

`$VT_INSTALL_PATH` is the path to the VampirTrace installation directory. It is not necessary to specify the VampirTrace include and library path, if it is installed in the default directory.

This uses automatic compiler instrumentation (`-finstrument-functions`) and the standard VampirTrace library. Replace the `-lvt` with `-lvt-mt` for multithreaded, `-lvt-mpi` for MPI and `-lvt-hyb` for multithreaded MPI applications. In this case the CUDA runtime library is linked before the `zlib`.

If the application is linked with `gcc/g++`, the linking command has to ensure, that the respective VampirTrace library is linked before the CUDA runtime library `libcudart.so` (check e.g. with "ldd *executable*"). Using the VampirTrace compiler wrappers (`vtcc/vtc++`) for linking is the easiest way to ensure correct linking of the VampirTrace library.

With the library tracing mechanism described in section 2.9, it is possible to trace CUDA applications without recompiling or relinking. There are only events written for Runtime API calls, kernels and communication between host and device.

Tracing the NVIDIA CUDA SDK 3.x and 4.0

To get some example traces, replace the compiler commands in the common Makefile include file (`common/common.mk`) with the corresponding VampirTrace compiler wrappers (⇒2.1) for automatic instrumentation:

```
# Compilers
NVCC := vtnvcc
CXX  := vtc++
CC   := vtcc
LINK := vtc++ #-vt:mt
```

Use the compiler switches for MPI, multithreaded and hybrid programs, if necessary (e.g. the CUDA SDK example `simpleMultiGPU` is a multithreaded program, which needs to be linked with a multithreaded VampirTrace library – uncomment the compiler switch in the linker command to use the multithreaded VampirTrace library).

Multithreaded CUDA applications

If threads are used to invoke asynchronous CUDA tasks, make sure to call a synchronizing CUDA function to get the tasks flushed before the thread exits. Otherwise tasks may not be flushed and will be missing in the trace file.

Mixed Use of CUDA runtime and driver API

As CUDA runtime API may implicitly create and destroy CUDA contexts, there might occur problems during CUDA event flushing. To workaround such an issue use only one API for interaction (memory copies, kernel execution) with the CUDA device. If you have to mix both APIs, make a clean exit for the API, which used the asynchronous tasks, before the other API closes its thread or context – cudaThreadExit() for runtime API and cuCtxDestroy() for driver API. Otherwise not yet flushed, asynchronous tasks will be missing in the final trace.

Note:
For 32-bit systems VampirTrace has to be configured with the 32-bit version of cuda runtime library. If the link test fails, use the following configure option (⇒A.2):

```
--with-cuda-lib-dir=$CUDA_INSTALL_PATH/lib
```

VampirTrace CUDA has been succesfully tested with the CUDA runtime version 3.x and 4.0.

4.6. Pthread API Calls

When tracing applications with Pthreads, only user events and functions are recorded which are automatically or manually instrumented. Pthread API functions will not be traced by default.
To enable tracing of all C-Pthread API functions include the header `vt_user.h` and compile the instrumented sources with `-DVTRACE_PTHREAD`.

C/C++:

```
#include "vt_user.h"
```

```
% vtcc -DVTRACE_PTHREAD hello.c -o hello
```

Note: Currently, Pthread instrumentation is only available for C/C++.

4.7. Plugin Counter Metrics

Plugin Counter add additional metrics to VampirTrace. They highly depend on the plugins, which are installed on your system. Every plugin should provide a README, which should be checked for available metrics. Once you have downloaded and compiled a plugin, copy the resulting library to a folder, which is part of your `LD_LIBRARY_PATH`. To enable the tracing of a specific metric, you should set the environment variable `VT_PLUGIN_CNTR_METRICS`. It is set in the following manner

```
export VT_PLUGIN_CNTR_METRICS=<library_name>_<event_name>
```

If you have for example a library named `libKswEvents.so` with the event `page_faults`, the you can set it with

```
export VT_PLUGIN_CNTR_METRICS=KswEvents_page_faults
```

Visit `http://www.tu-dresden.de/zih/vampirtrace/plugin_counter` for documentation and examples.

Note: Multiple events can be concatenated by using colons.

4.8. I/O Calls

Calls to functions which reside in external libraries can be intercepted by implementing identical functions and linking them before the external library. Such “wrapper functions” can record the parameters and return values of the library functions.

If VampirTrace has been built with I/O tracing support, it uses this technique for recording calls to I/O functions of the standard C library, which are executed by the application. The following functions are intercepted by VampirTrace:

`close`	`creat`	`creat64`	`dup`
`dup2`	`fclose`	`fcntl`	`fdopen`
`fgetc`	`fgets`	`flockfile`	`fopen`
`fopen64`	`fprintf`	`fputc`	`fputs`
`fread`	`fscanf`	`fseek`	`fseeko`
`fseeko64`	`fsetpos`	`fsetpos64`	`ftrylockfile`
`funlockfile`	`fwrite`	`getc`	`gets`
`lockf`	`lseek`	`lseek64`	`open`
`open64`	`pread`	`pread64`	`putc`
`puts`	`pwrite`	`pwrite64`	`read`
`readv`	`rewind`	`unlink`	`write`
`writev`			

The gathered information will be saved as I/O event records in the trace file. This feature has to be activated for each tracing run by setting the environment variable `VT_IOTRACE` to `yes`.

This works for both dynamically and statically linked executables. Note that when linking statically, a warning like the following may be issued: Using 'dlopen' in statically linked applications requires at runtime the shared libraries from the glibc version used for linking. This is ok as long as the mentioned libraries are available for running the application.

If you'd like to experiment with some other I/O library, set the environment variable `VT_IOLIB_PATHNAME` to the alternative one. Beware that this library must provide all I/O functions mentioned above otherwise VampirTrace will abort.

4.9. fork/system/exec Calls

If VampirTrace has been built with LIBC trace support (⇒ Appendix A), it is capable of tracing programs which call functions from the LIBC `exec` family (`execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`), `system`, and `fork`. VampirTrace records the call of the LIBC function to the trace. This feature works for sequential (i.e. no MPI or threaded parallelization) programs only. It works for both dynamically and statically linked executables. Note that when linking statically, a warning like the following may be issued: Using 'dlopen' in statically linked applications requires at runtime the shared libraries from the glibc version used for linking. This is ok as long as the mentioned libraries are available for running the application.

When VampirTrace detects a call of an `exec` function, the current trace file is closed before executing the new program. If the executed program is also instrumented with VampirTrace, it will create a different trace file. Note that VampirTrace aborts if the exec function returns unsuccessfully.

Calling `fork` in an instrumented program creates an additional process in the same trace file.

4.10. MPI Correctness Checking Using UniMCI

VampirTrace supports the recording of MPI correctness events, e.g., usage of invalid MPI requests. This is implemented by using the Universal MPI Correctness Interface (UniMCI), which provides an interface between tools like VampirTrace and existing runtime MPI correctness checking tools. Correctness events are stored as markers in the trace file and are visualized by Vampir.

If VampirTrace is built with UniMCI support, the user only has to enable MPI correctness checking. This is done by merely setting the environment variable `VT_MPICHECK` to `yes`. Further, if your application crashes due to an MPI error you should set `VT_MPICHECK_ERREXIT` to `yes`. This environmental variable forces VampirTrace to write its trace to disk and exit afterwards. As a result, the trace with the detected error is stored before the application might crash.

To install VampirTrace with correctness checking support it is necessary to have UniMCI installed on your system. UniMCI in turn requires you to have a supported MPI correctness checking tool installed, currently only the tool Marmot is known to have UniMCI support. So all in all you should use the following order to install with correctness checking support:

1. Marmot
   (see http://www.hlrs.de/organization/av/amt/research/marmot)
2. UniMCI
   (see http://www.tu-dresden.de/zih/unimci)
3. VampirTrace
   (see http://www.tu-dresden.de/zih/vampirtrace)

Information on how to install Marmot and UniMCI is given in their respective manuals. VampirTrace will automatically detect an UniMCI installation if the `unimci-config` tool is in path.

4.11. User-defined Counters

In addition to the manual instrumentation (⇒ Section 2.4), the VampirTrace API provides instrumentation calls which allow recording of program variable values (e.g. iteration counts, calculation results, ...) or any other numerical quantity. A user-defined counter is identified by its name, the counter group it belongs to, the type of its value (integer or floating-point) and the unit that the value is quoted (e.g. "GFlop/sec").

The VT_COUNT_GROUP_DEF and VT_COUNT_DEF instrumentation calls can be used to define counter groups and counters:

```
Fortran:
           #include "vt_user.inc"
           integer :: id, gid
           VT_COUNT_GROUP_DEF('name', gid)
           VT_COUNT_DEF('name', 'unit', type, gid, id)

C/C++:
           #include "vt_user.h"
           unsigned int id, gid;
           gid = VT_COUNT_GROUP_DEF("name");
           id = VT_COUNT_DEF("name", "unit", type, gid);
```

The definition of a counter group is optional. If no special counter group is desired, the default group "User" can be used. In this case, set the parameter gid of VT_COUNT_DEF() to VT_COUNT_DEFGROUP.

The third parameter type of VT_COUNT_DEF specifies the data type of the counter value. To record a value for any of the defined counters the corresponding instrumentation call VT_COUNT_*_VAL must be invoked.

Fortran:

Type	**Count call**	**Data type**
VT_COUNT_TYPE_INTEGER	VT_COUNT_INTEGER_VAL	integer (4 byte)
VT_COUNT_TYPE_INTEGER8	VT_COUNT_INTEGER8_VAL	integer (8 byte)
VT_COUNT_TYPE_REAL	VT_COUNT_REAL_VAL	real
VT_COUNT_TYPE_DOUBLE	VT_COUNT_DOUBLE_VAL	double precision

C/C++:

Type	**Count call**	**Data type**
VT_COUNT_TYPE_SIGNED	VT_COUNT_SIGNED_VAL	signed int (max. 64-bit)
VT_COUNT_TYPE_UNSIGNED	VT_COUNT_UNSIGNED_VAL	unsigned int (max. 64-bit)
VT_COUNT_TYPE_FLOAT	VT_COUNT_FLOAT_VAL	float
VT_COUNT_TYPE_DOUBLE	VT_COUNT_DOUBLE_VAL	double

The following example records the loop index `i`:

```
Fortran:

  #include "vt_user.inc"

  program main
  integer :: i, cid, cgid

  VT_COUNT_GROUP_DEF('loopindex', cgid)
  VT_COUNT_DEF('i', '#', VT_COUNT_TYPE_INTEGER, cgid, cid)

  do i=1,100
    VT_COUNT_INTEGER_VAL(cid, i)
  end do

  end program main


C/C++:

  #include "vt_user.h"

  int main() {
    unsigned int i, cid, cgid;

    cgid = VT_COUNT_GROUP_DEF('loopindex');
    cid = VT_COUNT_DEF("i", "#", VT_COUNT_TYPE_UNSIGNED,
                       cgid);

    for( i = 1; i <= 100; i++ ) {
      VT_COUNT_UNSIGNED_VAL(cid, i);
    }

    return 0;
  }
```

For all three languages the instrumented sources have to be compiled with `-DVTRACE`. Otherwise the `VT_*` calls are ignored.

Optionally, if the sources contain further VampirTrace API calls and only the calls for user-defined counters shall be disabled, then the sources have to be compiled with `-DVTRACE_NO_COUNT` in addition to `-DVTRACE`.

4.12. User-defined Markers

In addition to the manual instrumentation (⇒ Section 2.4), the VampirTrace API provides instrumentation calls which allow recording of special user information, which can be used to better identify parts of interest. A user-defined marker is identified by its name and type.

```
Fortran:
           #include "vt_user.inc"
           integer :: mid
           VT_MARKER_DEF('name', type, mid)
           VT_MARKER(mid, 'text')

C/C++:
           #include "vt_user.h"
           unsigned int mid;
           mid = VT_MARKER_DEF("name",type);
           VT_MARKER(mid, "text");

Types for Fortran/C/C++:
           VT_MARKER_TYPE_ERROR
           VT_MARKER_TYPE_WARNING
           VT_MARKER_TYPE_HINT
```

For all three languages the instrumented sources have to be compiled with `-DVTRACE`. Otherwise the `VT_*` calls are ignored.

Optionally, if the sources contain further VampirTrace API calls and only the calls for user-defined markers shall be disabled, then the sources have to be compiled with `-DVTRACE_NO_MARKER` in addition to `-DVTRACE`.

4.13. User-defined Communcation

In addition to the manual instrumentation (⇒ Section 2.4), the VampirTrace API provides instrumentation calls which allow recording of special user information, which can be used to better identify parts of interest. A user-defined communication operation is defined by a communicator and a tag. The default communicator is `VT_COMM_WORLD`. Additionally, a user-defined communicator can be created using `VT_COMM_DEF`:

```
Fortran:
         #include "vt_user.inc"
         integer :: cid
         VT_COMM_DEF('name', cid)
```

C/C++:
```
#include "vt_user.h"
unsigned cid;
cid = VT_COMM_DEF("name", cid);
```

Using VT_SEND and VT_RECV the user can insert send and receive events into the trace:

C/C++:
```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if( rank == 0 )
{
    for ( int i = 1; i < size; i++ )
    {
        VT_SEND(VT_COMM_WORLD,i,100);
    }
}else
{
    VT_RECV(VT_COMM_WORLD,rank,100);
}
```

The calls are similar for Fortran.

As can be seen, the arguments to VT_SEND and VT_RECV are a communicator, a tag and the size of the message. The tag is required in order to identify both ends of a user-defined communication. Therefore it has to be globally unique for a given communicator and cannot be reused within a single communicator. Messages with duplicated tags will not be visible in the final trace.

For all three languages the instrumented sources have to be compiled with -DVTRACE. Otherwise the VT_* calls are ignored. Optionally, if the sources contain further VampirTrace API calls and only the calls for user-defined markers shall be disabled, then the sources have to be compiled with -DVTRACE_NO_MSG in addition to -DVTRACE.

5. Filtering & Grouping

5.1. Function Filtering

By default, all calls of instrumented functions will be traced, so that the resulting trace files can easily become very large. In order to decrease the size of a trace, VampirTrace allows the specification of filter directives before running an instrumented application. The user can decide on how often an instrumented function/region shall be recorded to a trace file. To use a filter, the environment variable `VT_FILTER_SPEC` needs to be defined. It should contain the path and name of a file with filter directives.

Here is an example of a file containing filter directives:

```
#    VampirTrace region filter specification
#
#    call limit definitions and region assignments
#
#    syntax: <regions> -- <limit>
#
#      regions    semicolon-separated list of regions
#                 (can be wildcards)
#      limit      assigned call limit
#                  0 = region(s) denied
#                 -1 = unlimited
#
add;sub;mul;div -- 1000
* -- 3000000
```

These region filter directives cause that the functions `add`, `sub`, `mul` and `div` be recorded at most 1000 times. The remaining functions `*` will be recorded at most 3000000 times.

Besides creating filter files manually, you can also use the `vtfilter` tool to generate them automatically. This tool reads a provided trace and decides whether a function should be filtered or not, based on the evaluation of certain parameters. For more information see Section B.4.

Rank Specific Filtering

An experimental extension allows rank specific filtering. Use `@` clauses to restrict all following filters to the given ranks. The rank selection must be given as a list of `<from> - <to>` pairs or single values. Note that all rank specific rules are only effective after MPI_Init because the ranks is unknown before. The optional argument `-- OFF` disables the given ranks completely, regardless of following filter rules.

```
@ 35 - 42 -- OFF
@ 4 - 10, 20 - 29, 34
foo;bar -- 2000
* -- 0
```

The example defines two limits for the ranks 4 - 10, 20 - 29, and 34. The first line disables the ranks 35 - 42 completely.

Attention: The rank specific rules are activated later than usual at MPI_Init, because the ranks are not available earlier. The special MPI routines MPI_Init, MPI_Init_thread, and MPI_Initialized cannot be filtered in this way.

5.2. Java Specific Filtering

For Java tracing there are additional possibilities of filtering. Firstly, there is a default filter applied. The rules can be found in the filter file `<vt-install>/etc/vt-java-default-filter.spec`. Secondly, user-defined filters can be applied additionally by setting `VT_JAVA_FILTER_SPEC` to a file containing the rules.

The syntax of the filter rules is as follows:

```
<method|thread> <include|exclude> <filter string[;fs]...>
```

Filtering can be done on thread names and method names, defined by the first parameter. The second parameter determines whether the matching item shall be included for tracing or excluded from it. Multiple filter strings on a line have to be separated by `;` and may contain occurences of `*` for wildcard matching.

The user-supplied filter rules will be applied before the default filter and the first match counts so it is possible to include items that would be excluded by the default filter otherwise.

5.3. Function Grouping

VampirTrace allows assigning functions/regions to a group. Groups can, for instance, be highlighted by different colors in Vampir displays. The following standard groups are created by VampirTrace:

Group name	**Contained functions/regions**
MPI	MPI functions
OMP	OpenMP API function calls
OMP_SYNC	OpenMP barriers
OMP_PREG	OpenMP parallel regions
Pthreads	Pthread API function calls
MEM	Memory allocation functions (⇒ Section 4.3)
I/O	I/O functions (⇒ Section 4.8)
LIBC	LIBC fork/system/exec functions (⇒ Section 4.9)
Application	remaining instrumented functions and source code regions

Additionally, you can create your own groups, e.g., to better distinguish different phases of an application. To use function/region grouping set the environment variable VT_GROUPS_SPEC to the path of a file which contains the group assignments. Below, there is an example of how to use group assignments:

```
#     VampirTrace region groups specification
#
#     group definitions and region assignments
#
#     syntax: <group>=<regions>
#
#       group       group name
#       regions     semicolon-separated list of regions
#                   (can be wildcards)
#
CALC=add;sub;mul;div
USER=app_*
```

These group assignments associate the functions add, sub, mul and div with group "CALC", and all functions with the prefix app_ are associated with group "USER".

A. VampirTrace Installation

A.1. Basics

Building VampirTrace is typically a combination of running `configure` and `make`. Execute the following commands to install VampirTrace from the directory at the top of the tree:

```
% ./configure --prefix=/where/to/install
[...lots of output...]
% make all install
```

If you need special access for installing, you can execute `make all` as a user with write permissions in the build tree and a separate `make install` as a user with write permissions to the install tree.

However, for more details, also read the following instructions. Sometimes it might be necessary to provide `./configure` with options, e.g., specifications of paths or compilers.

VampirTrace comes with example programs written in C, C++, and Fortran. They can be used to test different instrumentation types of the VampirTrace installation. You can find them in the directory `examples` of the VampirTrace package.

Note that you should compile VampirTrace with the same compiler you use for the application to trace, see D.1.

A.2. Configure Options

Compilers and Options

Some systems require unusual options for compiling or linking which the `configure` script does not know. Run `./configure --help` for details on some of the pertinent environment variables.

You can pass initial values for configuration parameters to `configure` by setting variables in the command line or in the environment. Here is an example:

```
% ./configure CC=c89 CFLAGS=-O2 LIBS=-lposix
```

Installation Names

By default, `make install` will install the package's files in `/usr/local/bin`, `/usr/local/include`, etc. You can specify an installation prefix other than `/usr/local` by giving `configure` the option `--prefix=PATH`.

Optional Features

This a summary of the most important optional features. For a full list of all available features run `./configure --help`.

`--enable-compinst=TYPE`
enable support for compiler instrumentation, e.g. `gnu,pgi,pgi9,sun` default: automatically by configure. **Note:** Use `pgi9` for PGI compiler version 9.0 or higher.

`--enable-dyninst`
enable support for Dyninst instrumentation, default: enable if found by configure. **Note:** Requires Dyninst [1] version 6.1 or higher!

`--enable-dyninst-attlib`
build shared library which attaches Dyninst to the running application, default: enable if Dyninst found by configure and system supports shared libraries

`--enable-tauinst`
enable support for automatic source code instrumentation by using TAU, default: enable if found by configure. **Note:** Requires PDToolkit [2] or TAU [3]!

`--enable-memtrace`
enable memory tracing support, default: enable if found by configure

`--enable-cpuidtrace`
enable CPU ID tracing support, default: enable if found by configure

`--enable-libtrace=LIST`
enable library tracing support (gen,libc,io), default: automatically by configure

`--enable-rutrace`
enable resource usage tracing support, default: enable if found by configure

[1] http://www.dyninst.org
[2] http://www.cs.uoregon.edu/research/pdt/home.php
[3] http://tau.uoregon.edu

--enable-metrics=TYPE
enable support for hardware performance counter (`papi,cpc,necsx`), default: automatically by configure

--enable-zlib
enable ZLIB trace compression support, default: enable if found by configure

--enable-mpi
enable MPI support, default: enable if MPI found by configure

--enable-fmpi-lib
build the MPI Fortran support library, in case your system does not have a MPI Fortran library. default: enable if no MPI Fortran library found by configure

--enable-fmpi-handle-convert
do convert MPI handles, default: enable if MPI conversion functions found by configure

--enable-mpi2-thread
enable MPI-2 Thread support, default: enable if found by configure

--enable-mpi2-1sided
enable MPI-2 One-Sided Communication support, default: enable if found by configure

--enable-mpi2-extcoll
enable MPI-2 Extended Collective Operation support, default: enable if found by configure

--enable-mpi2-io
enable MPI-2 I/O support, default: enable if found configure

--enable-mpicheck
enable support for Universal MPI Correctness Interface (UniMCI), default: enable if unimci-config found by configure

--enable-etimesync
enable enhanced timer synchronization support, default: enable if C-LAPACK found by configure

--enable-threads=LIST
enable support for threads (pthread, omp), default: automatically by configure

--enable-java
enable Java support, default: enable if JVMTI found by configure

Important Optional Packages

This a summary of the most important optional features. For a full list of all available features run `./configure --help`.

`--with-platform=PLATFORM`
configure for given platform (`altix,bgl,bgp,crayt3e,crayx1,crayxt, ibm,linux,macos,necsx,origin,sicortex,sun,generic`), default: automatically by configure

`--with-bitmode=32|64`
specify bit mode

`--with-options=FILE`
load options from FILE, default: configure searches for a config file in config/defaults based on given platform and bitmode

`--with-local-tmp-dir=DIR`
give the path for node-local temporary directory to store local traces to, default: `/tmp`

If you would like to use an external version of OTF library, set:

`--with-extern-otf`
use external OTF library, default: not set

`--with-extern-otf-dir=OTFDIR`
give the path for OTF, default: `/usr`

`--with-otf-flags=FLAGS`
pass FLAGS to the OTF distribution configuration (only for internal OTF version)

`--with-otf-lib=OTFLIB`
use given otf lib, default: `-lotf -lz`

If the supplied OTF library was built without zlib support then OTFLIB will be set to `-lotf`.

`--with-dyninst-dir=DYNIDIR`
give the path for DYNINST, default: `/usr`

`--with-dyninst-inc-dir=DYNIINCDIR`
give the path for Dyninst-include files, default: `DYNIDIR/include`

`--with-dyninst-lib-dir=DYNILIBDIR`
give the path for Dyninst-libraries, default: `DYNIDIR/lib`

--with-dyninst-lib=DYNILIB
use given Dyninst lib, default: `-ldyninstAPI`

--with-tau-instrumentor=TAUINSTUMENTOR
give the command for the TAU instrumentor, default: `tau_instrumentor`

--with-pdt-cparse=PDTCPARSE
give the command for PDT C source code parser, default: `cparse`

--with-pdt-cxxparse=PDTCXXPARSE
give the command for PDT C++ source code parser, default: `cxxparse`

--with-pdt-fparse=PDTFPARSE
give the command for PDT Fortran source code parser, default: `f95parse, f90parse, or gfparse`

--with-papi-dir=PAPIDIR
give the path for PAPI, default: `/usr`

--with-cpc-dir=CPCDIR
give the path for CPC, default: `/usr`

If you have not specified the environment variable `MPICC` (MPI compiler command) use the following options to set the location of your MPI installation:

--with-mpi-dir=MPIDIR
give the path for MPI, default: `/usr/`

--with-mpi-inc-dir=MPIINCDIR
give the path for MPI-include files,
default: `MPIDIR/include/`

--with-mpi-lib-dir=MPILIBDIR
give the path for MPI-libraries, default: `MPIDIR/lib/`

--with-mpi-lib
use given mpi lib

--with-pmpi-lib
use given pmpi lib

If your system does not have an MPI Fortran library set `--enable-fmpi-lib` (see above), otherwise set:

--with-fmpi-lib
use given fmpi lib

Use the following options to specify your MPI-implementation

--with-hpmpi
set MPI-libs for HP MPI

--with-intelmpi
set MPI-libs for Intel MPI

--with-intelmpi2
set MPI-libs for Intel MPI2

--with-lam
set MPI-libs for LAM/MPI

--with-mpibgl
set MPI-libs for IBM BG/L

--with-mpibgp
set MPI-libs for IBM BG/P

--with-mpich
set MPI-libs for MPICH

--with-mpich2
set MPI-libs for MPICH2

--with-mvapich
set MPI-libs for MVAPICH

--with-mvapich2
set MPI-libs for MVAPICH2

--with-mpisx
set MPI-libs for NEC MPI/SX

--with-mpisx-ew
set MPI-libs for NEC MPI/SX with 8 Byte Fortran Integer

--with-openmpi
set MPI-libs for Open MPI

--with-sgimpt
set MPI-libs for SGI MPT

--with-sunmpi
set MPI-libs for SUN MPI

--with-sunmpi-mt
set MPI-libs for SUN MPI-MT

To enable enhanced timer synchronization a LAPACK library with C wrapper support is needed:

--with-clapack-dir=LAPACKDIR
set the path for CLAPACK, default: /usr

--with-clapack-lib
set CLAPACK-libs, default: -lclapack -lcblas -lf2c

--with-clapack-acml
set CLAPACK-libs for ACML

--with-clapack-essl
set CLAPACK-libs for ESSL

--with-clapack-mkl
set CLAPACK-libs for MKL

--with-clapack-sunperf
set CLAPACK-libs for SUN Performance Library

To enable Java support the JVM Tool Interface (JVMTI) version 1.0 or higher is required:

--with-jvmti-dir=JVMTIDIR
give the path for JVMTI, default: `$JAVA_HOME`

--with-jvmti-inc-dir=JVMTIINCDIR
give the path for JVMTI-include files, default: JVMTI/include

To enable support for generating wrapper for 3th-Party libraries the C code parser CTool is needed:

--with-ctool-dir=CTOOLDIR
give the path for CTool, default: /usr

--with-ctool-inc-dir=CTOOLINCDIR
give the path for CTool-include files, default: CTOOLDIR/include

--with-ctool-lib-dir=CTOOLLIBDIR
give the path for CTool-libraries, default: CTOOLDIR/lib

--with-ctool-lib=CTOOLLIB
use given CTool lib, default: automatically by configure

To enable support for CUDA runtime API wrapping, the CUDA-Toolkit install path is needed:

--with-cuda-dir=CUDATKDIR
give the path for CUDA Toolkit, default: /usr/local/cuda

--with-cuda-inc-dir=CUDATKINCDIR
give the path for CUDA Toolkit-include files, default: CUDATKDIR/include

--with-cuda-lib-dir=CUDATKLIBDIR
give the path for CUDA Toolkit-libraries, default: CUDATKDIR/lib64

--with-cudart-lib=CUDARTLIB
use given cudart lib, default: -lcudart

--with-cudart-shlib=CUDARTSHLIB
give the pathname for the shared CUDA runtime library, default: automatically by configure

To enable support for CUPTI counter capturing during CUDA runtime tracing, the CUPTI install path is needed:

--with-cupti-dir=CUPTIDIR
give the path for CUPTI, default: /usr

--with-cupti-inc-dir=CUPTIINCDIR
give the path for CUPTI-include files, default: CUPTIDIR/include

--with-cupti-lib-dir=CUPTILIBDIR
give the path for CUPTI-libraries, default: CUPTIDIR/lib

--with-cupti-lib=CUPTILIB
use given cupti lib, default: -lcupti

A.3. Cross Compilation

Building VampirTrace on cross compilation platforms needs some special attention. The compiler wrappers, OPARI, and the Library Wrapper Generator are built for the front-end (build system) whereas the the VampirTrace libraries, `vtdyn`, `vtunify`, and `vtfilter` are built for the back-end (host system). Some `configure` options which are of interest for cross compilation are shown below:

- Set `CC`, `CXX`, `F77`, and `FC` to the cross compilers installed on the front-end.
- Set `CC_FOR_BUILD` and `CXX_FOR_BUILD` to the native compilers of the front-end.
- Set `--host=` to the output of `config.guess` on the back-end.

- Set `--with-cross-prefix=` to a prefix which will be prepended to the executables of the compiler wrappers and OPARI (default: “cross-”)
- Maybe you also need to set additional commands and flags for the back-end (e.g. `RANLIB`, `AR`, `MPICC`, `CXXFLAGS`).

For example, this `configure` command line works for an NEC SX6 system with an X86_64 based front-end:

```
% ./configure CC=sxcc CXX=sxc++ F77=sxf90 FC=sxf90 MPICC=sxmpicc
              AR=sxar RANLIB="sxar st" CC_FOR_BUILD=cc CXX_FOR_BUILD=c++
              --host=sx6-nec-superux14.1
              --with-cross-prefix=sx
              --with-otf-lib=-lotf
```

A.4. Environment Set-Up

Add the `bin` subdirectory of the installation directory to your `$PATH` environment variable. To use VampirTrace with Dyninst, you will also need to add the lib subdirectory to your `LD_LIBRARY_PATH` environment variable:

for csh and tcsh:

```
> setenv PATH <vt-install>/bin:$PATH
> setenv LD_LIBRARY_PATH <vt-install>/lib:$LD_LIBRARY_PATH
```

for bash and sh:

```
% export PATH=<vt-install>/bin:$PATH
% export LD_LIBRARY_PATH=<vt-install>/lib:$LD_LIBRARY_PATH
```

A.5. Notes for Developers

Build from SVN

If you have checked out a *developer’s copy* of VampirTrace (i.e. checked out from CVS), you should first run:

```
% ./bootstrap [--otf-package <package>]
              [--version <version>]
```

Note that GNU Autoconf ≥2.60 and GNU Automake ≥1.9.6 are required. You can download them from http://www.gnu.org/software/autoconf and http://www.gnu.org/software/automake.

B. Command Reference

B.1. Compiler Wrappers (vtcc,vtcxx,vtf77,vtf90)

```
vtcc,vtcxx,vtf77,vtf90 - compiler wrappers for C, C++,
                         Fortran 77, Fortran 90

Syntax: vt<cc|cxx|f77|f90> [options] ...

options:
  -vt:help              Show this help message.
  -vt:version           Show VampirTrace version.
  -vt:<cc|cxx|f77|f90> <cmd>
                        Set the underlying compiler command.

  -vt:inst <insttype> Set the instrumentation type.

   possible values:

    compinst            fully-automatic by compiler
    manual              manual by using VampirTrace's API
    dyninst             binary by using Dyninst (www.dyninst.org)
    tauinst             automatic source code instrumentation by
                        using PDT/TAU

  -vt:opari <!args>     Set options for OPARI command. (see
                        share/vampirtrace/doc/opari/Readme.html)

  -vt:opari-rcfile <file>
                        Set pathname of the OPARI resource file.
                        (default: opari.rc)

  -vt:opari-table <file>
                        Set pathname of the OPARI runtime table file.
                        (default: opari.tab.c)

  -vt:noopari           Disable instrumentation of OpenMP contructs
                        by OPARI.

  -vt:<seq|mpi|mt|hyb>
```

```
                     Enforce application's parallelization type.
                     It's only necessary if it could not be determined
                     automatically based on underlying compiler and flags.
                     seq = sequential
                     mpi = parallel (uses MPI)
                     mt = parallel (uses OpenMP/POSIX threads)
                     hyb = hybrid parallel (MPI + Threads)
                     (default: automatically)

  -vt:tau <!args>    Set options for the TAU instrumentor
                     command.

  -vt:pdt <!args>    Set options for the PDT parse command.

  -vt:preprocess     Preprocess the source files before parsing
                     by OPARI and/or PDT.

  -vt:cpp <cmd>      Set C preprocessor command.

  -vt:cppflags <[!]flags>
                     Set/add flags for the C preprocessor.

  -vt:verbose        Enable verbose mode.

  -vt:show[me]       Do not invoke the underlying compiler.
                     Instead, show the command line that would be
                     executed to compile and link the program.

  -vt:showme-compile Do not invoke the underlying compiler.
                     Instead, show the compiler flags that would be
                     supplied to the compiler.

  -vt:showme-link    Do not invoke the underlying compiler.
                     Instead, show the linker flags that would be
                     supplied to the compiler.

  See the man page for your underlying compiler for other
  options that can be passed through 'vt<cc|cxx|f77|f90>'.

Environment variables:
  VT_INST            Equivalent to '-vt:inst'
  VT_CC              Equivalent to '-vt:cc '
  VT_CXX             Equivalent to '-vt:cxx '
  VT_F77             Equivalent to '-vt:f77'
  VT_F90             Equivalent to '-vt:f90'
  VT_CFLAGS          C compiler flags
```

```
  VT_CXXFLAGS         C++ compiler flags
  VT_F77FLAGS         Fortran 77 compiler flags
  VT_FCFLAGS          Fortran 90 compiler flags
  VT_LDFLAGS          Linker flags
  VT_LIBS             Libraries to pass to the linker

  The corresponding command line options overwrite the
  environment variables setting.

Examples:
  automatically instrumentation by compiler:

     vtcc -vt:cc gcc -vt:inst compinst -c foo.c -o foo.o
     vtcc -vt:cc gcc -vt:inst compinst -c bar.c -o bar.o
     vtcc -vt:cc gcc -vt:inst compinst foo.o bar.o -o foo

  manually instrumentation by using VT's API:

     vtf90 -vt:inst manual foobar.F90 -o foobar -DVTRACE

  IMPORTANT: Fortran source files instrumented by VT's API
             have to be preprocessed by CPP.
```

B.2. Local Trace Unifier (vtunify)

```
vtunify[-mpi] - local trace unifier for VampirTrace.

Syntax: vtunify[-mpi] <input trace prefix> [options]

options:
  -h, --help          Show this help message.

  -V, --version       Show VampirTrace version.

  -o PREFIX           Prefix of output trace filename.

  -f FILE             Function profile output filename.
                      (default=PREFIX.prof.txt)

  -k, --keeplocal     Don't remove input trace files.

  -p, --progress      Show progress.

  -v, --verbose       Increase output verbosity.
                      (can be used more than once)
```

`-q, --quiet`	`Enable quiet mode.` `(only emergency output)`
`--nocompress`	`Don't compress output trace files.`
`--nomsgmatch`	`Don't match messages.`
`--droprecvs`	`Drop message receive events, if msg. matching is enabled.`

B.3. Binary Instrumentor (vtdyn)

```
vtdyn - binary instrumentor (Dyninst mutator) for VampirTrace.

Syntax: vtdyn [options] <executable> [arguments ...]

options:
  -h, --help          Show this help message.

  -V, --version       Show VampirTrace version.

  -v, --verbose       Increase output verbosity.
                      (can be used more than once)

  -q, --quiet         Enable quiet mode.
                      (only emergency output)

  -o, --output FILE   Rewrite instrumented executable to specified pathname.

  -s, --shlibs SHLIBS[,...]
                      Comma-separated list of shared libraries which shall
                      also be instrumented.

  -f, --filter FILE   Pathname of input filter file.

  --ignore-nodbg      Don't instrument functions which have no debug
                      information.
```

B.4. Trace Filter Tool (vtfilter)

```
vtfilter[-mpi] - filter tool for VampirTrace.

Syntax:
  Generate a filter file:
    vtfilter[-mpi] --gen [gen-options] <input trace file>

  Filter a trace using an already existing filter file:
    vtfilter[-mpi] [--filt] [filt-options]
      --filter=<input filter file> <input trace file>

options:
  --gen                 Generate a filter file.
                        See 'gen-options' below for valid options.

  --filt                Filter a trace using an already existing
                        filter file. (default)
                        See 'filt-options' below for valid options.

  -h, --help            Show this help message.

  -V, --version         Show VampirTrace version.

  -p, --progress        Show progress.

  -v, --verbose         Increase output verbosity.
                        (can be used more than once)

gen-options:
  -o, --output=FILE     Pathname of output filter file.

  -r, --reduce=N        Reduce the trace size to N percent of the
                        original size. The program relies on the
                        fact that the major part of the trace are
                        function calls. The approximation of size
                        will get worse with a rising percentage of
                        communication and other non function
                        calling or performance counter records.

  -l, --limit=N         Limit the number of calls for filtered
                        function to N.
                        (default: 0)

  -s, --stats           Prints out the desired and the expected
                        percentage of file size.
```

```
  -e, --exclude=FUNC[;FUNC;...]
                        Exclude certain functions from filtering.
                        A function name may contain wildcards.

  --exclude-file=FILE   Pathname of file containing a list of
                        functions to be excluded from filtering.

  -i, --include=FUNC[;FUNC;...]
                        Force to include certain functions into
                        the filter. A function name may contain
                        wildcards.

  --include-file=FILE   Pathname of file containing a list of
                         functions to be included into the filter.

  --include-callees     Automatically include callees of included
                        functions as well into the filter.

filt-options:
  -o, --output=FILE     Pathname of output trace file.

  -f, --filter=FILE     Pathname of input filter file.

  -s, --max-streams=N   Maximum number of output streams.
                        (default: 0)
             vtfilter:  Set this to 0 to get the same number of
                        output streams as input streams.
         vtfilter-mpi:  Set this to 0 to get the same number of
                        output streams as MPI processes used, but
                        at least the number of input streams.

  --max-file-handles=N
                        Maximum number of files that are allowed
                        to be open simultaneously.
                        (default: 256)

  --nocompress          Don't compress output trace files.
```

B.5. Library Wrapper Generator (vtlibwrapgen)

```
vtlibwrapgen - library wrapper generator for VampirTrace.

Syntax:
  Generate a library wrapper source file:
    vtlibwrapgen [gen-options] <input header file>
                 [input header file...]

  Build a wrapper library from a generated source file:
    vtlibwrapgen --build [build-options]
                 <input lib. wrapper source file>

options:
  --gen              Generate a library wrapper source file.
                     This is the default behavior. See
                     'gen-options' below for valid options.

  --build            Build a wrapper library from a generated
                     source file. See 'build-options' below
                     for valid options.

  -h, --help         Show this help message.

  -V, --version      Show VampirTrace version.

  -q, --quiet        Enable quiet mode.
                     (only emergency output)

  -v, --verbose      Increase output verbosity.
                     (can be used more than once)

gen-options:
  -o, --output=FILE  Pathname of output wrapper source file.
                     (default: wrap.c)

  -l, --shlib=SHLIB  Pathname of shared library that contains
                     the actual library functions.
                     (can be used more then once)

  -f, --filter=FILE  Pathname of input filter file.

  -g, --group=NAME   Separate function group name for wrapped
                     functions.

  -s, --sysheader=FILE
```

```
                     Header file to be included additionally.

  --nocpp            Don't use preprocessor.

  --keepcppfile      Don't remove preprocessed header files.

  --cpp=CPP          C preprocessor command
                     (default: gcc -E)

  --cppflags=CPPFLAGS
                     C preprocessor flags, e.g.
                     -I<include dir>

  --cppdir=DIR       Change to this preprocessing directory.

environment variables:
  VT_CPP             C preprocessor command
                     (equivalent to '--cpp')
  VT_CPPFLAGS        C preprocessor flags
                     (equivalent to '--cppflags')

build-options:
  -o, --output=PREFIX
                     Prefix of output wrapper library.
                     (default: libwrap)

  --shared           Do only build shared wrapper library.

  --static           Do only build static wrapper library.

  --libtool=LT       Libtool command

  --cc=CC            C compiler command (default: gcc)

  --cflags=CFLAGS    C compiler flags

  --ld=LD            linker command (default: CC)

  --ldflags=LDFLAGS  linker flags, e.g. -L<lib dir>
                     (default: CFLAGS)

  --libs=LIBS        libraries to pass to the linker,
                     e.g. -l<library>

environment variables:
  VT_CC              C compiler command
```

```
                         (equivalent to '--cc')
  VT_CFLAGS              C compiler flags
                         (equivalent to '--cflags')
  VT_LD                  linker command
                         (equivalent to '--ld')
  VT_LDFLAGS             linker flags
                         (equivalent to '--ldflags')
  VT_LIBS                libraries to pass to the linker
                         (equivalent to '--libs')

examples:
  Generating wrapper library 'libm_wrap' for the Math library
  'libm.so':

    vtlibwrapgen -l libm.so -g MATH -o mwrap.c \
    /usr/include/math.h
    vtlibwrapgen --build -o libm_wrap mwrap.c
    export LD_PRELOAD=$PWD/libm_wrap.so:libvt.so
```

B.6. Application Execution Wrapper (vtrun)

```
 vtrun - application execution wrapper for VampirTrace.

 Syntax: vtrun [options] <executable> [arguments]

   options:
     -h, --help            Show this help message.

     -V, --version         Show VampirTrace version.

     -v, --verbose         Increase output verbosity.
                           (can be used more than once)

     -q, --quiet           Enable quiet mode.
                           (only emergency output)

     -<seq|mpi|mt|hyb>     Set application's parallelization type.
                           It's only necessary if it could not
                           be determined automatically.
                           seq = sequential
                           mpi = parallel (uses MPI)
                           mt  = parallel (uses OpenMP/POSIX threads)
                           hyb = hybrid parallel (MPI + Threads)
                           (default: automatically)
```

```
  --fortran           Set application's language to Fortran.
                      It's only necessary for MPI-applications
                      and if it could not be determined
                      automatically.

  --dyninst           Instrument user functions by Dyninst.

  --extra-libs=LIBS   Extra libraries to preload.

example:
  original:
     mpirun -np 4 ./a.out
  with VampirTrace:
     mpirun -np 4 vtrun ./a.out
```

C. Counter Specifications

C.1. PAPI

Available counter names can be queried with the PAPI commands `papi_avail` and `papi_native_avail`. Depending on the hardware there are limitations in the combination of different counters. To check whether your choice works properly, use the command `papi_event_chooser`.

```
PAPI_L[1|2|3]_[D|I|T]C[M|H|A|R|W]
              Level 1/2/3 data/instruction/total cache
              misses/hits/accesses/reads/writes

PAPI_L[1|2|3]_[LD|ST]M
              Level 1/2/3 load/store misses

PAPI_CA_SNP   Requests for a snoop
PAPI_CA_SHR   Requests for exclusive access to shared cache line
PAPI_CA_CLN   Requests for exclusive access to clean cache line
PAPI_CA_INV   Requests for cache line invalidation
PAPI_CA_ITV   Requests for cache line intervention

PAPI_BRU_IDL  Cycles branch units are idle
PAPI_FXU_IDL  Cycles integer units are idle
PAPI_FPU_IDL  Cycles floating point units are idle
PAPI_LSU_IDL  Cycles load/store units are idle

PAPI_TLB_DM   Data translation lookaside buffer misses
PAPI_TLB_IM   Instruction translation lookaside buffer misses
PAPI_TLB_TL   Total translation lookaside buffer misses

PAPI_BTAC_M   Branch target address cache misses
PAPI_PRF_DM   Data prefetch cache misses
PAPI_TLB_SD   Translation lookaside buffer shootdowns

PAPI_CSR_FAL  Failed store conditional instructions
PAPI_CSR_SUC  Successful store conditional instructions
PAPI_CSR_TOT  Total store conditional instructions

PAPI_MEM_SCY  Cycles Stalled Waiting for memory accesses
```

```
PAPI_MEM_RCY  Cycles Stalled Waiting for memory Reads
PAPI_MEM_WCY  Cycles Stalled Waiting for memory writes

PAPI_STL_ICY  Cycles with no instruction issue
PAPI_FUL_ICY  Cycles with maximum instruction issue
PAPI_STL_CCY  Cycles with no instructions completed
PAPI_FUL_CCY  Cycles with maximum instructions completed

PAPI_BR_UCN   Unconditional branch instructions
PAPI_BR_CN    Conditional branch instructions
PAPI_BR_TKN   Conditional branch instructions taken
PAPI_BR_NTK   Conditional branch instructions not taken
PAPI_BR_MSP   Conditional branch instructions mispredicted
PAPI_BR_PRC   Conditional branch instructions correctly
              predicted

PAPI_FMA_INS  FMA instructions completed
PAPI_TOT_IIS  Instructions issued
PAPI_TOT_INS  Instructions completed
PAPI_INT_INS  Integer instructions
PAPI_FP_INS   Floating point instructions
PAPI_LD_INS   Load instructions
PAPI_SR_INS   Store instructions
PAPI_BR_INS   Branch instructions
PAPI_VEC_INS  Vector/SIMD instructions
PAPI_LST_INS  Load/store instructions completed
PAPI_SYC_INS  Synchronization instructions completed
PAPI_FML_INS  Floating point multiply instructions
PAPI_FAD_INS  Floating point add instructions
PAPI_FDV_INS  Floating point divide instructions
PAPI_FSQ_INS  Floating point square root instructions
PAPI_FNV_INS  Floating point inverse instructions

PAPI_RES_STL  Cycles stalled on any resource
PAPI_FP_STAL  Cycles the FP unit(s) are stalled

PAPI_FP_OPS   Floating point operations
PAPI_TOT_CYC  Total cycles
PAPI_HW_INT   Hardware interrupts
```

C.2. CPC

Available counter names can be queried with the VampirTrace tool `vtcpcavail`. In addition to the counter names, it shows how many performance counters can be queried at a time. See below for a sample output.

```
% ./vtcpcavail
CPU performance counter interface: UltraSPARC T2
Number of concurrently readable performance counters
on the CPU: 2

Available events:
AES_busy_cycle
AES_op
Atomics
Br_completed
Br_taken
CPU_ifetch_to_PCX
CPU_ld_to_PCX
CPU_st_to_PCX
CRC_MPA_cksum
CRC_TCPIP_cksum
DC_miss
DES_3DES_busy_cycle
DES_3DES_op
DTLB_HWTW_miss_L2
DTLB_HWTW_ref_L2
DTLB_miss
IC_miss
ITLB_HWTW_miss_L2
ITLB_HWTW_ref_L2
ITLB_miss
Idle_strands
Instr_FGU_arithmetic
Instr_cnt
Instr_ld
Instr_other
Instr_st
Instr_sw
L2_dmiss_ld
L2_imiss
MA_busy_cycle
MA_op
MD5_SHA-1_SHA-256_busy_cycle
MD5_SHA-1_SHA-256_op
MMU_ld_to_PCX
RC4_busy_cycle
```

```
RC4_op
Stream_ld_to_PCX
Stream_st_to_PCX
TLB_miss

See the "UltraSPARC T2 User's Manual" for descriptions of these
events. Documentation for Sun processors can be found at:
http://www.sun.com/processors/manuals
```

C.3. NEC SX Hardware Performance Counter

This is a list of all supported hardware performance counters for NEC SX machines.

```
SX_CTR_STM     System timer reg
SX_CTR_USRCC   User clock counter
SX_CTR_EX      Execution counter
SX_CTR_VX      Vector execution counter
SX_CTR_VE      Vector element counter
SX_CTR_VECC    Vector execution clock counter
SX_CTR_VAREC   Vector arithmetic execution clock counter
SX_CTR_VLDEC   Vector load execution clock counter
SX_CTR_FPEC    Floating point data execution counter
SX_CTR_BCCC    Bank conflict clock counter
SX_CTR_ICMCC   Instruction cache miss clock counter
SX_CTR_OCMCC   Operand cache miss clock counter
SX_CTR_IPHCC   Instruction pipeline hold clock counter
SX_CTR_MNCCC   Memory network conflict clock counter
SX_CTR_SRACC   Shared resource access clock counter
SX_CTR_BREC    Branch execution counter
SX_CTR_BPFC    Branch prediction failure counter
```

C.4. Resource Usage

The list of resource usage counters can also be found in the manual page of `getrusage`. Note that, depending on the operating system, not all fields may be maintained. The fields supported by the Linux 2.6 kernel are shown in the table.

Name	Unit	Linux	Description
`ru_utime`	ms	x	Total amount of user time used.
`ru_stime`	ms	x	Total amount of system time used.
`ru_maxrss`	kB		Maximum resident set size.
`ru_ixrss`	kB $\times$ s		Integral shared memory size (text segment) over the runtime.
`ru_idrss`	kB $\times$ s		Integral data segment memory used over the runtime.
`ru_isrss`	kB $\times$ s		Integral stack memory used over the runtime.
`ru_minflt`	#	x	Number of soft page faults (i.e. those serviced by reclaiming a page from the list of pages awaiting reallocation).
`ru_majflt`	#	x	Number of hard page faults (i.e. those that required I/O).
`ru_nswap`	#		Number of times a process was swapped out of physical memory.
`ru_inblock`	#		Number of input operations via the file system. Note: This and `ru_oublock` do not include operations with the cache.
`ru_oublock`	#		Number of output operations via the file system.
`ru_msgsnd`	#		Number of IPC messages sent.
`ru_msgrcv`	#		Number of IPC messages received.
`ru_nsignals`	#		Number of signals delivered.
`ru_nvcsw`	#	x	Number of voluntary context switches, i.e. because the process gave up the processor before it had to (usually to wait for some resource to be available).
`ru_nivcsw`	#	x	Number of involuntary context switches, i.e. a higher priority process became runnable or the current process used up its time slice.

D. FAQ

D.1. Can I use different compilers for VampirTrace and my application?

There are several limitations which make this generally a bad idea:

- Using different compilers when tracing OpenMP applications does not work.
- Both compilers should have the same naming style for Fortran symbols (i.e. uppercase/lowercase, appending underscores) when tracing Fortran MPI applications.
- VampirTrace must be built to support the instrumentation type of the compiler you use for the application.

For example, the combination of a GCC compiled VampirTrace with an Intel compiled application will work except for OpenMP. But to avoid any trouble it is advisable to compile both VampirTrace and the application with the same compiler.

D.2. Why does my application need such a long time for starting?

If subroutines have been instrumented with automatic instrumentation by GNU, Intel, or PathScale compilers, VampirTrace needs to look-up the function names and their source code line before program start. In certain cases, this may take very long. To accelerate this process prepare a file with symbol information using the command `nm` as explained in Section 2.3 and set `VT_GNU_NMFILE` to the pathname of this file. This method prevents VampirTrace from getting the function names from the binary.

D.3. Why do I see multiple I/O operations for a single (un)formatted file read/write from my Fortran application?

VampirTrace does not implement any tracing at the Fortran language level. Therefore it is unaware of any I/O function calls done by Fortran applications.

However, if you enable I/O tracing using `VT_IOTRACE`, VampirTrace records all calls to LIBC's I/O functions. As Fortran uses the LIBC interface for executing its I/O operations, these function calls will be part of the trace. Depending on your Fortran compiler, a single Fortran file read/write operation may be split into several LIBC read calls which you will then see in your trace.

Beware that this may lead you to the (wrong) conclusion that your application spends time between the LIBC I/O calls inside the user function that contains the Fortran I/O call, especially when doing formatted I/O (see Figure D.1). It is rather the Fortran I/O subsystem which does all the formatting of the data that is eating your cpu cycles. But as this layer is unknown to VampirTrace, it cannot be shown and the time is accounted to the next higher function in the call stack - the user function.

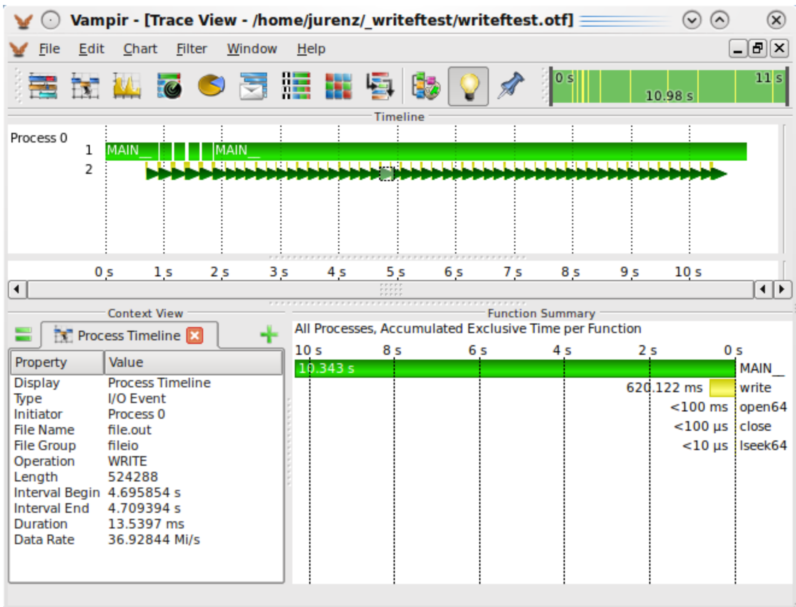

Figure D.1.: This trace of a Fortran application shows many isolated I/O operations and much time accounted to the MAIN function. Yet only a single formatted I/O write operation is issued in the code. As VampirTrace is not able to trace the Fortran I/O layer, it looks like the application itself uses cpu time between the traced LIBC I/O operations, which does not reflect the actual happenings.

D.4. The application has run to completion, but there is no *.otf file. What can I do?

The absence of an *.otf file usually means that the trace was not unified. This is the case on certain platforms, e.g. when using DYNINST or when the local traces are not available when the application ends and VampirTrace performs trace unification.

In those cases, a *.uctl file can be found in the directory of the trace file and the user needs to perform trace unification manually. See Sections 3.5 and B.2 to learn more about using vtunify.

D.5. What limitations are associated with "on/off" and buffer rewind?

Starting and stopping tracing by using the VT_ON/VT_OFF calls as well as the buffer rewind method are considered advanced usage of VampirTrace and should be performed with care. When restarting the recording of events, the call stack of the application has to have the same depth as when the recording was stopped. The same applies for the rewind call, which has to be at the same stack level as the rewind mark. If this is not the case, an error message will be printed during runtime and VampirTrace will abort execution. A safe method is to call VT_OFF and VT_ON in the same function.

It is allowed to use "on/off" in a section between a rewind mark and a buffer rewind call. But it is not allowed to call VT_SET_REWIND_MARK or VT_REWIND during a section deactivated by the "on/off" functionality.

Buffer flushes interfere with the rewind method: If the trace buffer is flushed after the call to VT_SET_REWIND_MARK, the mark is removed and a subsequent call to VT_REWIND will not work and issue a warning message.

In addition, stopping or rewinding tracing while waiting for MPI messages can cause those MPI messages not to be recorded in the trace. This can cause problems when analyzing the OTF trace afterwards, e.g., with Vampir.

D.6. VampirTrace warns that it "cannot lock file a.lock", what's wrong?

For unique naming of multiple trace files in the same directory, a file *.lock is created and locked for exclusive access if VT_FILE_UNIQUE is set to yes (⇒ Section 3.1). Some file systems do not implement file locking. In this case, VampirTrace still tries to name the trace files uniquely, but this may fail in certain cases. Alternatively, you can manually control the unique file naming by setting VT_FILE_UNIQUE to a different numerical ID for each program run.

D.7. Can I relocate my VampirTrace installation without rebuilding from source?

VampirTrace hard-codes some directory paths in its executables and libraries based on installation paths specified by the configure script. However, it's possible to move an existing VampirTrace installation to another location and use it without rebuild from source. Therefore it's necessary to set the environment variable VT_PREFIX to the new installation prefix before using VampirTrace's Compiler Wrappers (⇒ Section 2.1) or launching an instrumented application. For example:

```
./configure --prefix=/opt/vampirtrace
make install
mv /opt/vampirtrace $HOME/vampirtrace
export VT_PREFIX=$HOME/vampirtrace
```

D.8. What are the byte counts in collective communication records?

The byte counts in collective communication records changed with version 5.10.

From 5.10 on, the byte counts of collective communication records show the bytes per rank given to the MPI call or returned by the MPI call. This is the MPI API perspective. It is next to impossible to find out how many bytes are actually sent or received during a collective operation by any other MPI implementation.

In the past (until VampirTrace version 5.9), the byte count in collective operation records was defined differently. It used a simple and naive hypothetical implementation of collectives based on point-to-point messages and derived the byte counts from that. This might have been more confusing than helpful and was therefore changed.

Thanks to Eugene Loh for pointing this out!

D.9. I get "error: unknown asm constraint letter"

It is a known issue with the tau_instrumentor that it doesn't support inline assembler code. At the moment there is no other solution than using another kind of instrumentation like compiler instrumenation (⇒ Section 2.3) or manual instrumenation (⇒ Section 2.4).

D.10. I have a question that is not answered in this document!

You may contact us at vampirsupport@zih.tu-dresden.de for support on installing and using VampirTrace.

D.11. I need support for additional features so I can trace application xyz.

Suggestions are always welcome (contact: vampirsupport@zih.tu-dresden.de) but there is a chance that we can not implement all your wishes as our resources are limited.

Anyways, the source code of VampirTrace is open to everybody so you may implement support for new stuff yourself. If you provide us with your additions afterwards we will consider merging them into the official VampirTrace package.