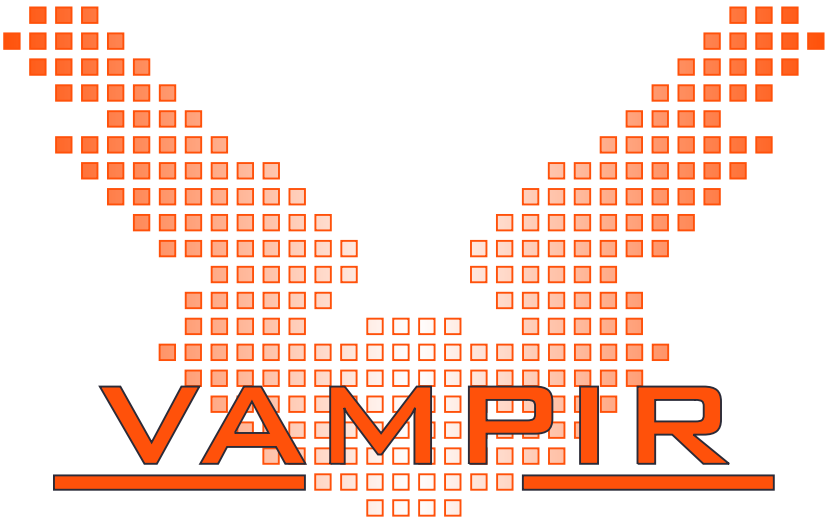
VampirTrace 5.4.9

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/

E-Mail: vampirsupport@zih.tu-dresden.de

Contents

This documentation describes how to prepare application programs in order to have traces generated, when executed. This step is called *instrumentation*. Furthermore, it explains how to control the run-time measurement system during execution (*tracing*). This also includes hardware performance counter sampling, as well as selective filtering and grouping of functions.

1 Introduction

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

The instrumentation part modifies a given application in order to inject additional measurement calls during run-time. The tracing part provides the current measurement functionality used by the instrumentation calls. By this means, a variety of detailed performance properties can be collected and recorded during run-time. This includes

- Function call enter and leave events
- MPI communication events
- OpenMP events
- Hardware performance counters
- various special purpose events

After a successful trace run, VampirTrace writes all collected data to a trace in the Open Trace Format (OTF), see http://www.tu-dresden.de/zih/otf.

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, see http://www.vampir.eu.

VampirTrace is included in Open MPI 1.3 and later. If not disabled explicitly, VampirTrace is built automatically when installing Open MPI. Refer to http://www.open-mpi.org/faq/?category=vampirtrace for more information.

Trace files can quickly become very large. With automatic instrumentation, even tracing applications that run only for 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. This produces trace files that are not larger than 32 MB per process, typically a lot smaller. Please read Section 3.2 on how to remove or change the limit.

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

2 Instrumentation

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

By default, VampirTrace handles this automatically. In order to enable 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 The Compiler Wrappers

All the necessary instrumentation of user functions as well as 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 and friends) with the extension of automatic instrumentation.

The following list shows some examples depending on the parallelization type of the program:

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

original:	`gfortran a.f90 b.f90 -o myprog`
with instrumentation:	`vtf90 a.f90 b.f90 -o myprog`

  This will instrument user functions (if supported by 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 MPI compilers (e.g. mpicc,

mpxlf90), you 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, you 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 (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).

- **OpenMP parallel programs**: When VampirTrace detects OpenMP flags on the command line, OPARI is invoked for automatic source code instrumentation of OpenMP events:

original:	`ifort -openmp pi.f -o pi`
with instrumentation:	`vtf77 -openmp pi.f -o pi`

For more information about OPARI refer to `share/vampirtrace/doc/opari/Readme.html` in VampirTrace's installation directory.

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

original:	`mpif90 -openmp hybrid.F90 -o hybrid`
with instrumentation:	`vtf90 -vt:f90 mpif90 -openmp hybrid.F90 -o hybrid`

The VampirTrace compiler wrappers try to detect automatically which parallelization method is used by means of the compiler flags (e.g. `-openmp` or `-lmpi`) 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 the switches `-vt:mpi`, `-vt:omp`, and `-vt:hyb` for MPI, OpenMP, 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 the compiler wrapper executes. Refer to Appendix A.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. For example, you could 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's option `-vt:inst <insttype>` specifies the instrumentation type to use. Following values for `<insttype>` are possible:

- fully-automatic instrumentation by the compiler (see Section 2.3):

insttype	**Compilers**
gnu	GNU (e.g., gcc, g++, gfortran, g95)
intel	Intel version $\geq$10.0 (e.g., icc, icpc, ifort)
pgi	Portland Group (PGI) (e.g., pgcc, pgCC, pgf90, pgf77)
phat	SUN Fortran 90 (e.g., cc, CC, f90)
xl	IBM (e.g., xlcc, xlCC, xlf90)
ftrace	NEC SX (e.g., sxcc, sxc++, sxf90)

- manual instrumentation (needs source-code modifications):

insttype	
manual	VampirTrace's API (see Section 2.4)
pomp	POMP INST directives (see Section 2.5)

- special instrumentation types (uses external tools):

insttype	
dyninst	binary-instrumentation with Dyninst (Section 2.6)

To determine which instrumentation type will be used by default and which other are available on your system take 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 Appendix A.1 or type `vtcc -vt:help` for other options that can be passed through VampirTrace's compiler wrapper.

2.3 Automatic Instrumentation

Automatic Instrumentation is the most convenient way to instrument your program. Simply use the compiler wrappers without any parameters, e.g.:

```
% vtf90 myprog1.f90 myprog2.f90 -o myprog
```

2.3.1 Notes for Using the GNU, Intel, or Pathscale Compiler

For these compilers the library *BFD* is required to get symbol information of the running application executable. This library is part of the *GNU Binutils*, which is downloadable from `http://www.gnu.org/software/binutils`.

To get the application executable for BFD 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 BFD.

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

```
% nm myprog > myprog.nm
```

Note that the output format of `nm` must be written in BSD-style. See the manual-page of nm for getting help about the output format setting.

2.3.2 Known License Issues with BFD

Please consider that BFD is deliviered under the GNU General Public License (GPL). So if you want to build binary packages including VampirTrace make sure to use the option `--without-bfd` to get a version without BFD. In this case you have to use `nm` to get symbol information from the running application executable (see Section 2.3.1).

2.3.3 Notes on Instrumentation of Inline Functions

Compilers behave differently when they automatically instrument inlined functions. The GNU and Intel $\geq$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 inlining enabled. Thus, one needs to disable inlining to enable instrumentation of inline functions and vice versa.

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

2.4 Manual Instrumentation using the VampirTrace API

The VT_USER_START, VT_USER_END instrumentation calls can be used to mark 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 by VT_USER_END, too.

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

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

For all three languages, the instrumented sources have to be compiled with -DVTRACE 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 instrumentation type with all other ones. For example, 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, like:

- without other instrumentation (e.g., compiler):

```
% vtcc -vt:inst manual myprog1.c -DVTRACE -o myprog
```

- combined with compiler-instrumentation:

```
% vtcc -vt:inst gnu myprog1.c -DVTRACE -o myprog
```

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

2.5 Manual Instrumentation using POMP

POMP (OpenMP Profiling Tool) instrumentation directives are supported for Fortran and C/C++. The main advantage is that by using directives, the instrumentation is ignored during normal compilation.

The `INST BEGIN` and `INST END` directives can be used to mark any user-defined sequence of statements. If this block has several exit points, all but the last exit point have to be instrumented by `INST ALTEND`.

```
Fortran:
          !POMP$ INST BEGIN(name)
          ...
            [ !POMP$ INST ALTEND(name) ]
          ...
          !POMP$ INST END(name)

C/C++:
          #pragma pomp inst begin(name)
          ...
            [ #pragma pomp inst altend(name) ]
          ...
          #pragma pomp inst end(name)
```

At least the main program function has to be instrumented in this way, and additionally, the following must be inserted as the first executable statement of the main program:

```
Fortran:
          !POMP$ INST INIT

C/C++:
          #pragma pomp inst init
```

2.6 Binary instrumentation using Dyninst

The option `-vt:inst dyninst` selects the compiler wrapper to instrument the application during run-time (binary instrumentation) by using Dyninst (http://www.dyninst.org). Recompiling is not necessary for this way of instrumenting, but relinking, as shown:

```
% vtf90 -vt:inst dyninst myprog1.o myprog2.o -o myprog
```

The compiler wrapper dynamically links the library `libvt.dynatt.so` to the application. This library attaches the *Mutator*-program `vtdyn` during run-time which invokes the instrumenting by using the Dyninst-API. Note that the application should have been compiled with the `-g` switch in order to have symbol names visible. After a trace-run by using this way of instrumenting, the `vtunify` utility needs to be invoked manually (see Sections 3.3 and A.2).

To prevent certain functions from being instrumented you can set the environment variable `VT_DYN_BLACKLIST` to a file containing a newline-separated list of function names. All additional overhead due to instrumentation of these functions will be removed.

VampirTrace also allows binary instrumentation of functions located in shared libraries. Ensure that the shared libraries have been compiled with `-g` and assign a colon-separated list of their names to the environment variable `VT_DYN_SHLIBS`, e.g.:

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

3 Runtime Measurement

By default, running a VampirTrace instrumented application should result in an OTF trace file in the current working directory where the application was executed. Use the environment variables `VT_FILE_PREFIX` and `VT_PFORM_GDIR` described below to change the name of the trace file and its final location. In case a problem occurs, set the environment variable `VT_VERBOSE` to `yes` before executing the instrumented application in order to see control messages of the VampirTrace run-time system which might help tracking down the problem.

The internal buffer of VampirTrace is limited to 32 MB. Use the environment variable `VT_BUFFER_SIZE` and `VT_MAX_FLUSHES` to increase this limit. Section 3.2 contains further information on influencing trace file size.

3.1 Environment Variables

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

Variable	Purpose	Default
`VT_PFORM_GDIR`	Name of global directory to store final trace file in	`./`
`VT_PFORM_LDIR`	Name of node-local directory that can be used to store temporary trace files	`/tmp/`
`VT_FILE_PREFIX`	Prefix used for trace filenames	`a`
`VT_APPPATH`	Path to the application executable	–
`VT_BUFFER_SIZE`	Size of internal event trace buffer. This is the place where event records are stored, before being written to a file.	32M
`VT_MAX_FLUSHES`	Maximum number of buffer flushes	1
`VT_VERBOSE`	Print VampirTrace related control information during measurement?	no
`VT_METRICS`	Specify counter metrics to be recorded with trace events as a colon-separated list of names. (for details see Appendix B)	–
`VT_MEMTRACE`	Enable memory allocation counters? (see Sec. 4.2)	no
`VT_IOTRACE`	Enable tracing of application I/O calls? (see Sec. 4.3)	no
`VT_MPITRACE`	Enable tracing of MPI events?	yes

VT_DYN_BLACKLIST	Name of blacklist file for Dyninst instrumentation (see Section 2.6)	–
VT_DYN_SHLIBS	Colon-separated list of shared libraries for Dyninst instrumentation (see Section 2.6)	–
VT_FILTER_SPEC	Name of function/region filter file (see Section 5.1)	–
VT_GROUPS_SPEC	Name of function grouping file (See Section 5.2)	–
VT_UNIFY	Unify local trace files afterwards?	yes
VT_COMPRESSION	Write compressed trace files?	yes

The value for the first three variables can 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 run-time.

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.

3.2 Influencing Trace File Size

The default values of the environment variables VT_BUFFER_SIZE and VT_MAX_FLUSHES limit the internal buffer of VampirTrace to 32 MB and the number of times that the buffer is flushed to 1. Events that should 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 the buffer is full. To change the size of the buffer, use the variable VT_BUFFER_SIZE. The optimal value for this variable depends on the application that should be traced. Setting a small value will increase the memory that is 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 this may cause parts of the application to be swapped to disk leading also to a significant change in the behavior of the application.

3.3 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. But under certain circumstances it is necessary to perform unification of local traces manually. To do this, use the command:

```
% vtunify <no-of-traces> <prefix>
```

For example, this is required on the BlueGene/L platform or when using Dyninst instrumentation.

4 Recording additional Events and Counters

4.1 PAPI Hardware Performance Counters

If VampirTrace has been built with hardware-counter support enabled (see Section C), VampirTrace is capable of recording hardware counter information as part of the event records. To request the measurement of certain counters, the user must set the environment variable `VT_METRICS`. The variable should contain a colon-separated list of counter names, or a predefined platform-specific group. Metric names can be any PAPI preset names or PAPI native counter names. For example, set

```
VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM
```

to record the number of floating point instructions and level 2 cache misses. See Appendix B for a full list of PAPI preset counters.

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.

4.2 Memory Allocation Counters

The GNU glibc implementation provides a special hook mechanism that allows intercepting all calls to 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 enabled (see Section C), 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 doesn't support memory tracing for OpenMP-parallelized programs!

4.3 Application 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. Following functions are intercepted by VampirTrace:

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

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`.

4.4 User Defined Counters

In addition to the manual instrumentation (see 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 optionally. 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. If additionally any functions or regions are manually instrumented by VT's API (see Section 2.4) and only the instrumentation calls for user defined counter should be disabled, then the sources have to be compiled with `-DVTRACE_NO_COUNT`, too.

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 is to 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.

Below, there 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` to be recorded at most 1000 times. The remaining functions `*` will be recorded at most 3000000 times.

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

5.2 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 constructs and functions
`MEM`	Memory allocation functions (see 4.2)
`I/O`	I/O functions (see 4.3)
`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 make the functions `add`, `sub`, `mul` and `div` associated with group “CALC” and all functions with the prefix `app_` are associated with group “USER”.

A Command Reference

A.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> [-vt:<cc|cxx|f77|f90> <cmd>]
        [-vt:inst <insttype>] [-vt:<seq|mpi|omp|hyb>]
        [-vt:opari <args>] [-vt:verbose] [-vt:version]
        [-vt:showme] [-vt:showme_compile]
        [-vt:showme_link] ...

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

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

   possible values:

     gnu              fully-automatic by GNU compiler
     intel            ... Intel (version >= 10.x) ...
     pgi              ... Portland Group (PGI) ...
     phat             ... SUN Fortran 90 ...
     xl               ... IBM ...
     ftrace           ... NEC SX ...
     manual           manual by using VampirTrace's API
     pomp             manual by using using POMP INST directives
     dyninst          binary by using Dyninst (www.dyninst.org)

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

  -vt:<seq|mpi|omp|hyb>
                      Force application's parallelization type.
                      Necessary, if this cannot be determined
                      by underlying compiler and flags.
                      seq = sequential
```

```
                        mpi = parallel (uses MPI)
                        omp = parallel (uses OpenMP)
                        hyb = hybrid parallel (MPI + OpenMP)
                        (default: automatically determining by
                         underlying compiler and flags)

  -vt:verbose           Enable verbose mode.

  -vt:showme            Do not invoke the underlying compiler.
                        Instead, show the command line that
                        would be executed.

  -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_CC                 Equivalent to '-vt:cc'
  VT_CXX                Equivalent to '-vt:cxx'
  VT_F77                Equivalent to '-vt:f77'
  VT_F90                Equivalent to '-vt:f90'
  VT_INST               Equivalent to '-vt:inst'

  The corresponding command line options overwrite the
  environment variable settings.

Examples:
  automatically instrumentation by using GNU compiler:

     vtcc -vt:cc gcc -vt:inst gnu -c foo.c -o foo.o
     vtcc -vt:cc gcc -vt:inst gnu -c bar.c -o bar.o
     vtcc -vt:cc gcc -vt:inst gnu 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 or
             POMP directives have to be preprocessed by CPP.
```

A.2 Local Trace Unifier (vtunify)

```
vtunify - local trace unifier for VampirTrace.

Syntax: vtunify <#files> <iprefix> [-o <oprefix>]
        [-c|--compress <on|off>] [-k|--keeplocal]
        [-v|--verbose]

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

  #files               number of local trace files
                       (equal to # of '*.uctl' files)

  iprefix              prefix of input trace filename.

  -o <oprefix>         prefix of output trace filename.

  -s <statsofile>      statistics output filename
                       default=<oprefix>.stats

  -q, --noshowstats    Don't show statistics on stdout.

  -c, --nocompress     Don't compress output trace files.

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

  -v, --verbose        Enable verbose mode.
```

A.3 Dyninst Mutator (vtdyn)

```
vtdyn - Dyninst Mutator for VampirTrace.

Syntax: vtdyn [-v|--verbose] [-s|--shlib <shlib>[,...]]
              [-b|--blacklist <bfile> [-p|--pid <pid>]
              <app> [appargs ...]

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

  -v, --verbose        Enable verbose mode.

  -s, --shlib          Comma-separated list of shared libraries
  <shlib>[,...]        which should also be instrumented.

  -b, --blacklist      Set path of blacklist file containing
  <bfile>              a newline-separated list of functions
                       which should not be instrumented.

  -p, --pid <pid>      application's process id
                       (attaches the mutator to a running process)

  app                  path of application executable

  appargs              application's arguments
```

A.4 Trace Filter Tool (vtfilter)

```
vtfilter - filter generator for VampirTrace

Syntax:
    Filter a trace file using an already existing filter file:
       vtfilter -filt [filt-options] <input trace file>
    Generate a filter:
       vtfilter -gen [gen-options] <input trace file>

general options:
    -h, --help              show this help message
    -p                      show progress

filt-options:
    -to <file>              output trace file name

    -fi <file>              input filter file name

    -z <zlevel>             Set the compression level. Level
                            reaches from 0 to 9 where 0 is no
                            compression and 9 is the highest
                            level. Standard is 4.

    -f <n>                  Set max number of file handles
                            available. Standard is 256.

gen-options:
    -fo <file>              output filter file name

    -r <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 <n>                  Limit the number of accepted
                            function calls for filtered functions
                            to <n>. Standard is 0.

    -ex <f>,<f>,...         Exclude certain symbols from
                            filtering. A symbol may contain
```

```
                        wildcards.

   -in <f>,<f>,...      Force to include certain symbols
                        into the filter. A symbol may contain
                        wildcards.

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

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


environment variables:
  TRACEFILTER_EXCLUDEFILE  Specifies a file containing a list
                           of symbols not to be filtered. The
                           list of members can be seperated
                           by space, comma, tab, newline and
                           may contain wildcards.

  TRACEFILTER_INCLUDEFILE  Specifies a file containing a list
                           of symbols  to be filtered.
```

B PAPI Counter Specifications

Available counter names can be queried with the PAPI commands `papi_avail` and `papi_native_avail`. There are limitations to the combinations of 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 VampirTrace Installation

C.1 Basics

Building VampirTrace is typically a combination of running `configure` and `make`. Execute the following commands to install VampirTrace from within 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, then 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. Please consult the CONFIG-EXAMPLES file to get an idea of how to configure VampirTrace for your platform.

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.

C.2 Configure Options

Compilers and Options

Some systems require unusual options for compiling or linking that the `configure` script does not know about. 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

`--enable-compinst=COMPINSTLIST`
enable support for compiler instrumentation,
e.g. (`gnu,intel,pgi,phat,xl,ftrace`),
A VampirTrace installation can handle different compilers.
The first item in the list is the run-time default.
default: automatically by configure

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

`--enable-omp`
enable OpenMP support, default: enable if compiler supports OpenMP

`--enable-hyb`
enable Hybrid (MPI/OpenMP) support, default: enable if MPI found and compiler supports OpenMP

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

`--enable-iotrace`
enable libc's I/O tracing support, default: enable if libdl found by configure

`--enable-dyninst`
enable support for Dyninst instrumentation,
default: enable if found by configure
Note: Requires Dyninst version 5.0.1 or higher!
(`http://www.dyninst.org`)

`--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-papi`
enable PAPI hardware counter support,
default: enable if 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

Important Optional Packages

--with-local-tmp-dir=LTMPDIR
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/local/`

--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 used 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/local/`

--with-papi-dir=PAPIDIR
give the path for PAPI, 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

C.3 Cross Compilation

Building VampirTrace on cross compilation platforms needs some special attention. The compiler wrappers and OPARI are built for the front-end (build system) whereas 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 `CXX_FOR_BUILD` to the native compiler of the front-end (used to compile compiler wrappers and OPARI only).
- Set `--host=` to the output of `config.guess` on the back-end.
- 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" CXX_FOR_BUILD=c++
              --host=sx6-nec-superux14.1
              --with-otf-lib=-lotf
```

C.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
```

C.5 Notes for Developers

Build from CVS

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

```
% ./bootstrap
```

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

Creating a distribution tarball (VampirTrace-X.X.X.tar.gz)

If you would like to create a new distribution tarball, run:

```
% ./makedist -o <otftarball> <major> <minor> <release>
```

instead of `make dist`. The script `makedist` adapts the version number `<major>.<minor>.<release>` in `configure.in` and extracts given OTF-tarball `<otftarball>` in `./extlib/otf/`.