Extrae

User guide manual
for version 2.2.1

Harald Servat Gelabert & Germán Llort Sánchez
tools@bsc.es

March 6, 2012

Contents

List of Tables

Chapter 1

Quick start guide

1.1 The instrumentation package

Extrae is a dynamic instrumentation package to trace programs compiled and run with the shared memory model (like OpenMP and pthreads), the message passing (MPI) programming model or both programming models (different MPI processes using OpenMP or pthrads within each MPI process). Extrae generates trace files that can be latter visualized with Paraver .

The package is distributed in compressed tar format (e.g., extrae.tar.gz). To unpack it, execute from the desired target directory the following command line :

```
gunzip -c extrae.tar.gz | tar -xvf -
```

The unpacking process will create different directories on the current directory (see table 1.1).

Directory	Contents
bin	Contains the binary files of the Extrae tool.
etc	Contains some scripts to set up environment variables and the Extrae internal files.
lib	Contains the Extrae tool libraries.
share/man	Contains the Extrae manual entries.
share/doc	Contains the Extrae manuals (pdf, ps and html versions).
share/example	Contains examples to illustrate the Extrae instrumentation.

Table 1.1: Package contents description

1.2 Quick running

Once the package has been unpacked, set the `EXTRAE_HOME` environment variable to the directory where the package was installed. Use the `export` or `setenv` commands to set it up depending on the shell you use. If you use sh-based shell (like sh, bash, ksh, zsh, ...), issue this command

```
export EXTRAE_HOME=dir
```

however, if you use csh-based shell (like csh, tcsh), execute the following command

```
setenv EXTRAE_HOME dir
```

where *dir* refers where the Extrae was installed. Henceforth, all references to the usage of the environment variables will be used following the sh format unless specified.

Extrae is offered in two different flavors: as a DynInst-based application, or stand-alone application. DynInst is a dynamic instrumentation library that allows the injection of code in a running application without the need to recompile the target application. If the DynInst instrumentation library is not installed, Extrae also offers different mechanisms to trace applications.

1.2.1 Quick running Extrae - based on DynInst

Extrae needs some environment variables to be setup on each session. Issuing the command

```
source ${EXTRAE_HOME}/etc/extrae.sh
```

on a sh-based shell, or

```
source ${EXTRAE_HOME}/etc/extrae.csh
```

on a csh-based shell will do the work. Then copy the default XML configuration file[1] into the working directory by executing

```
cp ${EXTRAE_HOME}/share/example/MPI/extrae.xml .
```

If needed, set the application environment variables as usual (like `OMP_NUM_THREADS`, for example), and finally launch the application using the `${EXTRAE_HOME}/bin/extrae` instrumenter like:

```
${EXTRAE_HOME}/bin/extrae -config extrae.xml <program>
```

where `<program>` is the application binary.

1.2.2 Quick running Extrae - NOT based on DynInst

Extrae needs some environment variables to be setup on each session. Issuing the command

```
source ${EXTRAE_HOME}/etc/extrae.sh
```

on a sh-based shell, or

```
source ${EXTRAE_HOME}/etc/extrae.csh
```

on a csh-based shell will do the work. Then copy the default XML configuration file[1] into the working directory by executing

```
cp ${EXTRAE_HOME}/share/example/MPI/extrae.xml .
```

and export the EXTRAE_CONFIG_FILE as

[1] See section 4 for further details regarding this file

```
export EXTRAE_CONFIG_FILE=extrae.xml
```

If needed, set the application environment variables as usual (like `OMP_NUM_THREADS`, for example). Just before executing the target application, issue the following command:

```
export LD_PRELOAD=${EXTRAE_HOME}/lib/<lib>
```

where `<lib>` is

- `libmpitrace.so` If the application is just MPI.
- `libomptrace.so` If the application is just OpenMP.
- `libompitrace.so` If the application is MPI and OpenMP.

1.3 Quick merging the intermediate traces

Once the intermediate trace files (*.mpit files) have been created, they have to be merged (using the `mpi2prv` command) in order to generate the final Paraver trace file. Execute the following command to proceed with the merge:

```
${EXTRAE_HOME}/bin/mpi2prv -f TRACE.mpits -o output.prv
```

The result of the merge process is a Paraver tracefile called output.prv. If the -o option is not given, the resulting tracefile is called EXTRAE_Paraver_Trace.prv.

Chapter 2

Introduction

Extrae is a dynamic instrumentation package to trace programs compiled and run with the shared memory model (like OpenMP and pthreads), the message passing (MPI) programming model or both programming models (different MPI processes using OpenMP or pthreads within each MPI process). Extrae generates trace files that can be visualized with Paraver .

Extrae is currently available on different platforms and operating systems: IBM PowerPC running Linux or AIX, and x86 and x86-64 running Linux. It also has been ported to OpenSolaris and FreeBSD.

The combined use of Extrae and Paraver offers an enormous analysis potential, both qualitative and quantitative. With these tools the actual performance bottlenecks of parallel applications can be identified. The microscopic view of the program behavior that the tools provide is very useful to optimize the parallel program performance.

This document tries to give the basic knowledge to use the Extrae tool. Chapter 3 explains how the package can be configured and installed. Chapter 7 explains how to monitor an application to obtain its trace file. At the end of this document there are appendices that include: a Frequent Asked Questions appendix and a list of routines instrumented in the package.

What is the Paraver tool?

Paraver is a flexible parallel program visualization and analysis tool based on an easy-to-use Motif GUI. Paraver was developed responding to the need of hacing a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Paraver provides a large amount of information useful to decide the points on which to invest the programming effort to optimize an application.

Expressive power, flexibility and the capability of efficiently handling large traces are key features addressed in the design of Paraver . The clear and modular structure of Paraver plays a significant role towers achieving these targets.

Some Paraver features are the support for:

- Detailed quantitative analysis of program performance,
- concurrent comparative analysis of several traces,
- fast analysis of very large traces,
- support for mixed message passing and shared memory (network of SMPs), and,

- customizable semantics of the visualized information.

One of the main features of Paraver is the flexibility to represent traces coming from different environments. Traces are composed of state records, events and communications with associated timestamp. These three elements can be used to build traces that capture the behavior along time of very different kind of systems. The Paraver distribution includes, either in its own distribution or as additional packages, the following instrumentation modules:

1. Sequential application tracing: it is included in the Paraver distribution. It can be used to trace the value of certain variables, procedure invocations, ... in a sequential program.

2. Parallel application tracing: a set of modules are optionally available to capture the activity of parallel applications using shared-memory, message-passing paradigms, or a combination of them.

3. System activity tracing in a multiprogrammed environment: an application to trace processor allocations and process migrations is optionally available in the Paraver distribution.

4. Hardware counters tracing: an application to trace the hardware counter values is optionally available in the Paraver distribution.

Where the Paraver tool can be found?

The Paraver distribution can be found at URL:

`http://www.bsc.es/paraver`

Paraver binaries are available for Linux/x86, Linux/x86-64 and Linux/ia64, Windows.

In the Documentation Tool section of the aforementioned URL you can find the Paraver *Reference Manual* and Paraver *Tutorial* in addition to the documentation for other instrumentation packages.

Extrae and Paraver tools e-mail support is **tools@bsc.es**.

Chapter 3

Configuration, build and installation

3.1 Configuration of the instrumentation package

There are many options to be applied at configuration time for the instrumentation package. We point out here some of the relevant options, sorted alphabetically. To get the whole list run `configure --help`. Options can be enabled or disabled. To enable them use –enable-X or –with-X= (depending on which option is available), to disable them use –disable-X or –without-X.

- `--enable-merge-in-trace`
  Embed the merging process in the tracing library so the final tracefile can be generated automatically from the application run.

- `--enable-parallel-merge`
  Build the parallel mergers (mpimpi2prv/mpimpi2dim) based on MPI.

- `--enable-posix-clock`
  Use POSIX clock (clock_gettime call) instead of low level timing routines. Use this option if the system where you install the instrumentation package modifies the frequency of its processors at runtime.

- `--enable-single-mpi-lib`
  Produces a single instrumentation library for MPI that contains both Fortran and C wrappers. Applications that call the MPI library from both C and Fortran languages need this flag to be enabled.

- `--enable-spu-write`
  Enable direct write operations to disk from SPUs in CELL machines avoiding the usage of DMA transfers. The write mechanism is very slow compared with the original behavior.

- `--enable-sampling`
  Enable PAPI sampling support.

- `--enable-pmapi`
  Enable PMAPI library to gather CPU performance counters. PMAPI is a base package installed in AIX systems since version 5.2.

- `--enable-cuda`
  Enable support for tracing CUDA calls on nVidia hardware. This instrumentation is only valid in binaries that use the shared version of the CUDA library. Interposition has to be done through the `LD_PRELOAD` mechanism. It is superseded by `--with-cupti` which also supports instrmentation for static binaries.

- `--enable-openmp`
  Enable support for tracing OpenMP on IBM and GNU runtimes. The IBM runtime instrumentation is only available for PowerPC systems.

- `--enable-smpss`
  Enable support for tracing SMP-superscalar.

- `--enable-nanos`
  Enable support for tracing Nanos run-time.

- `--enable-pthread`
  Enable support for tracing pthread library calls.

- `--enable-upc`
  Enable support for tracing UPC run-time.

- `--enable-xml`
  Enable support for XML configuration (not available on BG/L, BG/P and BG/Q systems).

- `--enable-xmltest`
  Do not try to compile and run a test LIBXML program.

- `--enable-doc`
  Generates this documentation.

- `--prefix=DIR`
  Location where the installation will be placed. After issuing `make install` you will find under DIR the entries `lib/`, `include/`, `share/` and `bin/` containing everything needed to run the instrumentation package.

- `--with-mpi=DIR`
  Specify the location of an MPI installation to be used for the instrumentation package. This flag is mandatory.

- `--with-binary-type=OPTION`
  Available options are: 32, 64 and default. Specifies the type of memory address model when compiling (32bit or 64bit).

- `--with-mpi-name-mangling=OPTION`
  Available options are: 0u, 1u, 2u, upcase and auto. Choose the Fortran name decoration (0, 1 or 2 underscores) for MPI symbols. Let OPTION be auto to automatically detect the name mangling.

- `--with-pacx=DIR`
  Specify where PACX communication library can be find.

- `--with-unwind=DIR`
  Specify where to find Unwind libraries and includes. This library is used to get callstack information on several architectures (including IA64 and Intel x86-64). This flag is mandatory.

- `--with-papi=DIR`
  Specify where to find PAPI libraries and includes. PAPI is used to gather performance counters. This flag is mandatory.

- `--with-bfd=DIR`
  Specify where to find the Binary File Descriptor package. In conjunction with libiberty, it is used to translate addresses into source code locations.

- `--with-liberty=DIR`
  Specify where to find the libiberty package. In conjunction with Binary File Descriptor, it is used to translate addresses into source code locations.

- `--with-dyninst=DIR`
  Specify the installation location for the DynInst package. Extrae also requires the DWARF package `--with-dwarf=DIR` when using DynInst. This flag is mandatory.

- `--with-cupti=DIR`
  Specify the location of the CUPTI libraries. CUPTI is used to instrument CUDA calls, and supersedes the `--enable-cuda`.

3.2 Build

To build the instrumentation package, just issue `make` after the configuration.

3.3 Installation

To install the instrumentation package in the directory chosen at configure step (through `--prefix` option), issue `make install`.

3.4 Examples of configuration on different machines

All commands given here are given as an example to configure and install the package, you may need to tune them properly (i.e., choose the appropriate directories for packages and so). These examples assume that you are using a sh/bash shell, you must adequate them if you use other shells (like csh/tcsh).

3.4.1 Bluegene (L and P variants)

Configuration command:

```
./configure --prefix=/homec/jzam11/jzam1128/aplic/extrae/2.2.0
--with-papi=/homec/jzam11/jzam1128/aplic/papi/4.1.2.1
--with-bfd=/bgsys/local/gcc/gnu-linux_4.3.2/powerpc-linux-gnu/powerpc-bgp-linux
--with-liberty=/bgsys/local/gcc/gnu-linux_4.3.2/powerpc-bgp-linux
--with-mpi=/bgsys/drivers/ppcfloor/comm --without-unwind --without-dyninst
```

Build and installation commands:

```
make
make install
```

3.4.2 BlueGene/Q

Configuration command:

```
./configure --prefix=/bgsys/home/zrlrod/pub/release/bgq/gnu/extrae
--with-mpi=/bgsys/drivers/ppcfloor/comm/gcc --without-unwind
--without-dyninst --without-papi --disable-openmp --disable-pthread
--with-libz=no
```

Build and installation commands:

```
make
make install
```

3.4.3 AIX

Some extensions of Extrae do not work properly (nanos, SMPss and OpenMP) on AIX. In addition, if using IBM MPI (aka POE) the make will complain when generating the parallel merge if the main compiler is not xlc/xlC. So, you can either change the compiler or disable the parallel merge at compile step. Also, command `ar` can complain if 64bit binaries are generated. It's a good idea to run make with OBJECT_MODE=64 set to avoid this.

Compiling the 32bit package using the IBM compilers

Configuration command:

```
CC=xlc CXX=xlC ./configure --prefix=PREFIX --disable-nanos --disable-smpss
--disable-openmp --with-binary-type=32 --without-unwind --enable-pmapi
--without-dyninst --with-mpi=/usr/lpp/ppe.poe
```

Build and installation commands:

```
make
make install
```

Compiling the 64bit package without the parallel merge

Configuration command:

```
./configure --prefix=PREFIX --disable-nanos --disable-smpss --disable-openmp
--disable-parallel-merge --with-binary-type=64 --without-unwind
--enable-pmapi --without-dyninst --with-mpi=/usr/lpp/ppe.poe
```

Build and installation commands:

```
OBJECT_MODE=64 make
make install
```

3.4.4 Linux

Compiling using default binary type using MPICH, OpenMP and PAPI

Configuration command:

```
./configure --prefix=PREFIX --with-mpi=/home/harald/aplic/mpich/1.2.7
--with-papi=/usr/local/papi --enable-openmp --without-dyninst
--without-unwind
```

Build and installation commands:

```
make
make install
```

Compiling 32bit package in a 32/64bit mixed environment

Configuration command:

```
./configure --prefix=PREFIX --with-mpi=/opt/osshpc/mpich-mx
--with-papi=/gpfs/apps/PAPI/3.6.2-970mp --with-binary-type=32
--with-unwind=$HOME/aplic/unwind/1.0.1/32 --with-elf=/usr --with-dwarf=/usr
--with-dyninst=$HOME/aplic/dyninst/7.0.1/32
```

Build and installation commands:

```
make
make install
```

Compiling 64bit package in a 32/64bit mixed environment

Configuration command:

```
./configure --prefix=PREFIX --with-mpi=/opt/osshpc/mpich-mx
--with-papi=/gpfs/apps/PAPI/3.6.2-970mp --with-binary-type=64
--with-unwind=$HOME/aplic/unwind/1.0.1/64 --with-elf=/usr --with-dwarf=/usr
--with-dyninst=$HOME/aplic/dyninst/7.0.1/64
```

Build and installation commands:

```
make
make install
```

Compiling using default binary type using OpenMPI and PACX

Configuration command:

```
./configure --prefix=PREFIX --with-mpi=/home/harald/aplic/openmpi/1.3.1
--with-pacx=/home/harald/aplic/pacx/07.2009-openmpi --without-papi
--without-unwind --without-dyninst
```

Build and installation commands:

```
make
make install
```

Compiling using default binary type, using OpenMPI, DynInst and libunwind

Configuration command:

```
./configure --prefix=PREFIX --with-mpi=/home/harald/aplic/openmpi/1.3.1
--with-dyninst=/home/harald/dyninst/7.0.1 --with-dwarf=/usr
--with-elf=/usr --with-unwind=/home/harald/aplic/unwind/1.0.1
--without-papi
```

Build and installation commands:

```
make
make install
```

Compiling on CRAY XT5 for 64bit package and adding sampling

Notice the "–disable-xmltest". As backends programs cannot be run in the frontend, we skip running the XML test. Also using a local installation of libunwind.

Configuration command:

```
CC=cc CFLAGS='-O3 -g' LDFLAGS='-O3 -g' CXX=CC CXXFLAGS='-O3 -g' ./configure
--with-mpi=/opt/cray/mpt/4.0.0/xt/seastar/mpich2-gnu --with-binary-type=64
--with-xml-prefix=/sw/xt5/libxml2/2.7.6/sles10.1_gnu4.1.2
--disable-xmltest --with-bfd=/opt/cray/cce/7.1.5/cray-binutils
--with-liberty=/opt/cray/cce/7.1.5/cray-binutils --enable-sampling
--enable-shared=no --prefix=PREFIX --with-papi=/opt/xt-tools/papi/3.7.2/v23
--with-unwind=/ccs/home/user/lib --without-dyninst
```

Build and installation commands:

```
make
make install
```

Compiling on a Power CELL processor using Linux

Configuration command:

```
./configure --with-mpi=/opt/openmpi/ppc32 --without-unwind --without-dyninst
--without-papi --prefix=/gpfs/data/apps/CEPBATOOLS/extrae/2.2.0/openmpi/32
--with-binary-type=32
```

Build and installation commands:

```
make
make install
```

Compiling on a ARM based processor machine using Linux

If using the GNU toolchain to compile the library, we suggest at least using version 4.6.2 because of its enhaced in this architecture.

Configuration command:

```
CC=/gpfs/APPS/BIN/GCC-4.6.2/bin/gcc-4.6.2 ./configure
--prefix=/gpfs/CEPBATOOLS/extrae/2.2.0 --with-unwind=/gpfs/CEPBATOOLS/libunwind/1.0.1-g
--with-papi=/gpfs/CEPBATOOLS/papi/4.2.0 --with-mpi=/usr --enable-posix-clock
--without-dyninst
```

Build and installation commands:

```
make
make install
```

Compiling in a Slurm/MOAB environment with support for MPICH2

Configuration command:

```
export MP_IMPL=anl2 ./configure --prefix=PREFIX
--with-mpi=/gpfs/apps/MPICH2/mx/1.0.8p1..3/32
--with-papi=/gpfs/apps/PAPI/3.6.2-970mp --with-binary-type=64
--without-dyninst --without-unwind
```

Build and installation commands:

```
make
make install
```

Compiling in a environment with IBM compilers and POE

Configuration command:

```
CC=xlc CXX=xlC ./configure --prefix=PREFIX --with-mpi=/opt/ibmhpc/ppe.poe
--without-dyninst --without--unwind --without-papi
```

Build and installation commands:

```
make
make install
```

Compiling in a environment with GNU compilers and POE

Configuration command:

```
./configure --prefix=PREFIX --with-mpi=/opt/ibmhpc/ppe.poe --without-dyninst
--without--unwind --without-papi
```

Build and installation commands:

```
MP_COMPILER=gcc make
make install
```

3.5 Knowing how a package was configured

If you are interested on knowing how an Extrae package was configured execute the following command after setting EXTRAE_HOME to the base location of an installation

```
${EXTRAE_HOME}/etc/configured.sh
```

this command will show the configure command itself and the location of some dependencies of the instrumentation package.

Chapter 4

Extrae XML configuration file

Extrae is configured through a XML file that is set through the `EXTRAE_CONFIG_FILE` environment variable. The included examples provide several XML files to serve as a basis for the end user. There are four XML files:

- `extrae.xml` Exemplifies all the options available to set up in the configuration file. We will discuss below all the sections and options available. It is also available on this document on appendix A.
- `extrae_explained.xml` The same as the above with some comments on each section.
- `summarized_trace_basic.xml` A small example for gathering information of MPI and OpenMP information with some performace counters and calling information at each MPI call.
- `detailed_trace_basic.xml` A small example for gathering a summarized information of MPI and OpenMP parallel paradigms.

Please note that most of the nodes present in the XML file have an `enabled` attribute that allows turning on and off some parts of the instrumentation mechanism. For example, `<mpi enabled="yes">` means MPI instrumentation is enabled and process all the contained XML subnodes, if any; whether `<mpi enabled="no">` means to skip gathering MPI information and do not process XML subnodes.

Each section points which environment variables could be used if the tracing package lacks XML support. See appendix B for the entire list.

Sometimes the XML tags are used for time selection (duration, for instance). In such tags, the following postfixes can be used: n for nanoseconds, u for microseconds, m for milliseconds, s for seconds, M for minutes, H for hours and D for days.

4.1 XML Section: Trace configuration

The basic trace behavior is determined in the first part of the XML and **contains** all of the remaining options. It looks like:

```
<?xml version='1.0'?>
```

```
<trace enabled="yes"
 home="@sed_MYPREFIXDIR@"
 initial-mode="detail"
 type="paraver"
 xml-parser-id="@sed_XMLID@"
>

< ... other XML nodes ... >

</trace>
```

The `<?xml version='1.0'?>` is mandatory for all XML files. Don't touch this. The available tunable options are under the `<trace>` node:

- `enabled` Set to `"yes"` if you want to generate tracefiles.
- `home` Set to where the instrumentation package is installed. Usually it points to the same location that `EXTRAE_HOME` environment variable.
- `initial-mode` Available options
  - `detail` Provides detailed information of the tracing.
  - `bursts` Provides summarized information of the tracing. This mode removes most of the information present in the detailed traces (like OpenMP and MPI calls among others) and only produces information for computation bursts.
- `type` Available options
  - `paraver` The intermediate files are meant to generate Paraver tracefiles.
  - `dimemas` The intermediate files are meant to generate Dimemas tracefiles.
- `xml-parser-id` This is used to check whether the XML parsing scheme and the file scheme match or not.

See **EXTRAE_ON**, **EXTRAE_HOME**, **EXTRAE_INITIAL_MODE** and **EXTRAE_TRACE_TYPE** environment variables in appendix B.

4.2 XML Section: MPI

The MPI configuration part is nested in the config file (see section 4.1) and its nodes are the following:

```
<mpi enabled="yes">
  <counters enabled="yes" />
</mpi>
```

MPI calls can gather performance information at the begin and end of MPI calls. To activate this behavior, just set to yes the attribute of the nested `<counters>` node.

> See **EXTRAE_DISABLE_MPI** and **EXTRAE_MPI_COUNTERS_ON** environment variables in appendix B.

4.3 XML Section: PACX

The PACX configuration part is nested in the config file (see section 4.1) and its nodes are the following:

```
<pacx enabled="yes">
  <counters enabled="yes" />
</pacx>
```

PACX calls can gather performance information at the begin and end of PACX calls. To activate this behavior, just set to yes the attribute of the nested `<counters>` node.

> See **EXTRAE_DISABLE_PACX** and **EXTRAE_PACX_COUNTERS_ON** environment variables in appendix B.

4.4 XML Section: pthread

The pthread configuration part is nested in the config file (see section 4.1) and its nodes are the following:

```
<pthread enabled="yes">
  <locks enabled="no" />
  <counters enabled="yes" />
</pthread>
```

The tracing package allows to gather information of some pthread routines. In addition to that, the user can also enable gathering information of locks and also gathering performance counters in all of these routines. This is achieved by modifying the enabled attribute of the `<locks>` and `<counters>`, respectively.

> See **EXTRAE_DISABLE_PTHREAD**, **EXTRAE_PTHREAD_LOCKS** and **EXTRAE_PTHREAD_COUNTERS_ON** environment variables in appendix B.

4.5 XML Section: OpenMP

The OpenMP configuration part is nested in the config file (see section 4.1) and its nodes are the following:

```
<openmp enabled="yes">
  <locks enabled="no" />
  <counters enabled="yes" />
</openmp>
```

The tracing package allows to gather information of some OpenMP runtimes and outlined routines. In addition to that, the user can also enable gathering information of locks and also gathering performance counters in all of these routines. This is achieved by modifying the enabled attribute of the `<locks>` and `<counters>`, respectively.

> See **EXTRAE_DISABLE_OMP**, **EXTRAE_OMP_LOCKS** and **EXTRAE_OMP_COUNTERS_ON** environment variables in appendix B.

4.6 XML Section: CELL

The Cell configuration part is only parsed for tracing packages suited for the Cell architecture, and as the rest of sections it is nested in the config file (see section 4.1). The available nodes only affect the SPE side, and they are:

```
<cell enabled="no">
  <spu-file-size enabled="yes">5</spu-file-size>
  <spu-buffer-size enabled="yes">64</spu-buffer-size>
  <spu-dma-channel enabled="yes">2</spu-dma-channel>
</cell>
```

- `spu-file-size` Limits the resulting intermediate trace file for each SPE thread that has been instrumented.
- `spu-buffer-size` Specifies the number of events contained in the buffer on the SPE side. Remember that memory is very scarce on the SPE, so setting a high value can exhaust all memory.
- `spu-dma-channel` Chooses which DMA channel will be used to perform the intermediate trace files transfers to the PPE side.

> See **EXTRAE_SPU_FILE_SIZE**, **EXTRAE_SPU_BUFFER_SIZE** and **EXTRAE_SPU_DMA_CHANNEL** environment variables in appendix B.

4.7 XML Section: Callers

```
<callers enabled="yes">
  <mpi enabled="yes">1-3</mpi>
  <pacx enabled="no">1-3</pacx>
  <sampling enabled="no">1-5</sampling>
</callers>
```

Callers are the routine addresses present in the process stack at any given moment during the application run. Callers can be used to link the tracefile with the source code of the application.

The instrumentation library can collect a partial view of those addresses during the instrumentation. Such collected addresses are translated by the merging process if the correspondent parameter is given and the application has been compiled and linked with debug information.

There are three points where the instrumentation can gather this information:

- Entry of MPI calls
- Entry of PACX calls
- Sampling points *(if sampling is available in the tracing package)*

The user can choose which addresses to save in the trace (starting from 1, which is the closest point to the MPI call or sampling point) specifying several stack levels by separating them by commas or using the hyphen symbol.

See **EXTRAE_MPI_CALLER** and **EXTRAE_PACX_CALLER** environment variables in appendix B.

4.8 XML Section: User functions

```
<user-functions enabled="no" list="/home/bsc41/bsc41273/user-functions.dat">
  <counters enabled="yes" />
</user-functions>
```

There are two different mechanisms to instrument user functions. One is using DynInst support, the other relies on compiling and linking the application using additional options.

- If you use DynInst support in the instrumentation package, then the pointed list in the `list` attribute within the tag is just a name list of the functions to be traced.
- If you use the IBM XL compilers, specify the option `-qdebug=function_trace` at compile and link stages. The `list` attribute, as in DynInst, points to a name list of functions to be traced.
- If you use the GNU C compiler with the option `-finstrument-functions` at compile and link stages, the `list` attribute must point a file with a list of entries like:

  `function_name # address`

  You can generate this list by using the nm command applied to the binary. For example, `nm | grep FUNCTION` will show you the function name, followed by the type of FUNCTION (should be T or t) and then followed by the address of the symbol.

Finally, in order to gather performance counters in these functions and also in those instrumented using the `extrae_user_function` API call, the node `counters` has to be enabled.

See **EXTRAE_FUNCTIONS** environment variable in appendix B.

4.9 XML Section: Performance counters

The instrumentation library can be compiled with support for collecting performance metrics of different components available on the system. These components include:

- Processor performance counters. Such access is granted by PAPI[1] or PMAPI[2]
- Network performance counters. *(Only available in systems with Myrinet GM/MX networks).*
- Operating system accounts.

Here is an example of the counters section in the XML configuration file:

```
<counters enabled="yes">
  <cpu enabled="yes" starting-set-distribution="1">
    <set enabled="yes" domain="all" changeat-time="5s">
      PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM
      <sampling enabled="yes" period="100000000">PAPI_TOT_CYC</sampling>
    </set>
    <set enabled="yes" domain="user" changeat-globalops="5">
      PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_FP_INS
    </set>
  </cpu>
  <network enabled="yes" />
  <resource-usage enabled="yes" />
</counters>
```

See **EXTRAE_COUNTERS**, **EXTRAE_NETWORK_COUNTERS** and **EXTRAE_RUSAGE** environment variables in appendix B.

4.9.1 Processor performance counters

Processor performance counters are configured in the `<cpu>` nodes. The user can configure many sets in the `<cpu>` node using the `<set>` node, but just one set will be used at any given time in a specific task. The `<cpu>` node supports the `tarting-set-distribution` attribute with the following accepted values:

- `number` (*in range 1..N, where N is the number of configured sets*) All tasks will start using the set specified by number.
- `block` Each task will start using the given sets distributed in blocks (*i.e.*, if two sets are defined and there are four running tasks: tasks 1 and 2 will use set 1, and tasks 3 and 4 will use set 2).

[1] More information available on their website `http://icl.cs.utk.edu/papi`. Extrae requires PAPI 3.x at least.

[2] PMAPI is only available for AIX operating system, and it is on the base operating system since AIX5.3. Extrae requires AIX 5.3 at least.

- `cyclic` Each task will start using the given sets distributed cyclically (*i.e.*, if two sets are defined and there are four running tasks: tasks 1 and 3 will use, and tasks 2 and 4 will use set 2).

Each set contain a list of performance counters to be gathered at different instrumentation points (see sections 4.2, 4.5 and 4.8). If the tracing library is compiled to support PAPI, performance counters must be given using the canonical name (like PAPI_TOT_CYC and PAPI_L1_DCM), or the PAPI code in hexadecimal format (like 8000003b and 80000000, respectively)[3]. If the tracing library is compiled to support PMAPI, only one group identifier can be given per set[4] and can be either the group name (like pm_basic and pm_hpmcount1) or the group number (like 6 and 22, respectively).

In the given example (which refers to PAPI support in the tracing library) two sets are defined. First set will read PAPI_TOT_INS (total instructions), PAPI_TOT_CYC (total cycles) and PAPI_L1_DCM (1st level cache misses). Second set is configured to obtain PAPI_TOT_INS (total instructions), PAPI_TOT_CYC (total cycles) and PAPI_FP_INS (floating point instructions).

Additionally, if the underlying performance library supports sampling mechanisms, each set can be configured to gather information (see section 4.7) each time the specified counter reaches a specific value. The counter that is used for sampling must be present in the set. In the given example, the first set is enabled to gather sampling information every 100M cycles.

Furthermore, performance counters can be configured to report accounting on different basis depending on the `domain` attribute specified on each set. Available options are

- `kernel` Only counts events ocurred when the application is running in kernel mode.
- `user` Only counts events ocurred when the application is running in user-space mode.
- `all` Counts events independently of the application running mode.

In the given example, first set is configured to count all the events ocurred, while the second one only counts those events ocurred when the application is running in user-space mode.

Finally, the instrumentation can change the active set in a manual and an automatic fashion. To change the active set manually see `Extrae_previous_hwc_set` and `Extrae_next_hwc_set` API calls in 5.1. To change automatically the active set two options are allowed: based on time and based on application code. The former mechanism requires adding the attribute `changeat-time` and specify the minimum time to hold the set. The latter requires adding the attribute `changeat-globalops` with a value. The tracing library will automatically change the active set when the application has executed as many MPI global operations as selected in that attribute. When In any case, if either attribute is set to zero, then the set will not me changed automatically.

4.9.2 Network performance counters

Network performance counters are only available on systems with Myrinet GM/MX networks and they are fixed depending on the firmware used. Other systems, like BG/* may provide some network performance counters, but they are accessed through the PAPI interface (see section 4.9 and PAPI documentation).

If `<network>` is enabled the network performance counters appear at the end of the application run, giving a summary for the whole run.

[3]Some architectures do not allow grouping some performance counters in the same set.

[4]Each group contains several performance counters

4.9.3 Operating system accounting

Operating system accounting is obtained through the getrusage(2) system call when `<resource-usage>` is enabled. As network performance counters, they appear at the end of the application run, giving a summary for the whole run.

4.10 XML Section: Storage management

The instrumentation packages can be instructed on what/where/how produce the intermediate trace files. These are the available options:

```
<storage enabled="no">
  <trace-prefix enabled="yes">TRACE</trace-prefix>
  <size enabled="no">5</size>
  <temporal-directory enabled="yes">/scratch</temporal-directory>
  <final-directory enabled="yes">/gpfs/scratch/bsc41/bsc41273</final-directory>
  <gather-mpits enabled="no" />
</storage>
```

Such options refer to:

- `trace-prefix` Sets the intermediate trace file prefix. Its default value is TRACE.
- `size` Let the user restrict the maximum size (in megabytes) of each resulting intermediate trace file[5].
- `temporal-directory` Where the intermediate trace files will be stored during the execution of the application. By default they are stored in the current directory. If the directory does not exist, the instrumentation will try to make it.
- `final-directory` Where the intermediate trace files will be stored once the execution has been finished. By default they are stored in the current directory. If the directory does not exist, the instrumentation will try to make it.
- `gather-mpits` If the system does not provide a global filesystem the resulting trace files will be distributed among the computation nodes. Turning on this option will use the underlying communication mechanism (MPI) to gather all the intermediate trace files into the root node.

See **EXTRAE_PROGRAM_NAME**, **EXTRAE_FILE_SIZE**, **EXTRAE_DIR**, **EXTRAE_FINAL_DIR** and **EXTRAE_GATHER_MPITS** environment variables in appendix B.

[5]This check is done each time the buffer is flushed, so the resulting size of the intermediate trace file depends also on the number of elements contained in the tracing buffer (see section 4.11).

4.11 XML Section: Buffer management

Modify the buffer management entry to tune the tracing buffer behavior.

```
<buffer enabled="yes">
  <size enabled="yes">150000</size>
  <circular enabled="no" />
</buffer>
```

By, default (even if the enabled attribute is "no") the tracing buffer is set to 500k events (see section 4.6 for further information of buffer in the CELL). If `<size>` is enabled the tracing buffer will be set to the number of events indicated by this node. If the circular option is enabled, the buffer will be created as a circular buffer and the buffer will be dumped only once with the last events generated by the tracing package.

See **EXTRAE_BUFFER_SIZE** environment variable in appendix B.

4.12 XML Section: Trace control

```
<trace-control enabled="yes">
  <file enabled="no" frequency="5M">/gpfs/scratch/bsc41/bsc41273/control</file>
  <global-ops enabled="no">10</global-ops>
  <remote-control enabled="yes">
    <mrnet enabled="yes" target="150" analysis="spectral" start-after="30">
      <clustering max_tasks="26" max_points="8000"/>
      <spectral min_seen="1" max_periods="0" num_iters="3" signals="DurBurst,InMPI"/>
    </mrnet>
    <signal enabled="no" which="USR1"/>
  </remote-control>
</trace-control>
```

This section groups together a set of options to limit/reduce the final trace size. There are three mechanisms which are based on file existance, global operations executed and external remote control procedures.

Regarding the `file`, the application starts with the tracing disabled, and it is turned on when a control file is created. Use the property `frequency` to choose at which frequency this check must be done. If not supplied, it will be checked every 100 global operations on MPI_COMM_WORLD.

If the `global-ops` tag is enabled, the instrumentation package begins disabled and starts the tracing when the given number of global operations on MPI_COMM_WORLD has been executed.

The `remote-control` tag section allows to configure some external mechanisms to automatically control the tracing. Currently, there is only one option which is built on top of MRNet and it is based on clustering and spectral analysis to generate a small yet representative trace.

These are the options in the `mrnet` tag:

- **target**: the approximate requested size for the final trace (in Mb).

- **analysis**: one between `clustering` and `spectral`.
- **start-after**: number of seconds before the first analysis starts.

The `clustering` tag configures the clustering analysis parameters:

- **max_tasks**: maximum number of tasks to get samples from.
- **max_points**: maximum number of points to cluster.

The `spectral` tag section configures the spectral analysis parameters:

- **min_seen**: minimum times a given type of period has to be seen to trace a sample
- **max_periods**: maximum number of representative periods to trace. 0 equals to unlimited.
- **num_iters**: number of iterations to trace for every representative period found.
- signals: performance signals used to analyze the application. If not specified, `DurBurst` is used by default.

A signal can be used to terminate the tracing when using the remote control. Available values can be only USR1/USR2 Some MPI implementations handle one of those, so check first which is available to you. Set in tag `signal` the signal code you want to use.

> See **EXTRAE_CONTROL_FILE**, **EXTRAE_CONTROL_GLOPS**, **EXTRAE_CONTROL_TIME** environment variables in appendix B.

4.13 XML Section: Bursts

```
<bursts enabled="no">
  <threshold enabled="yes">500u</threshold>
  <mpi-statistics enabled="yes" />
  <pacx-statistics enabled="yes" />
</bursts>
```

If the user enables this option, the instrumentation library will just emit information of computation bursts (*i.e.*, not does not trace MPI calls, OpenMP runtime, and so on) when the current mode (through initial-mode in 4.1) is set to `bursts`. The library will discard all those computation bursts that last less than the selected threshold.

In addition to that, when the tracing library is running in burst mode, it computes some statistics of MPI and PACX activity. Such statistics can be dumped in the tracefile by enabling `mpi-statistics` and `pacx-statistics` respectively.

> See **EXTRAE_INITIAL_MODE**, **EXTRAE_BURST_THRESHOLD**, **EXTRAE_MPI_STATISTICS** and **EXTRAE_PACX_STATISTICS** environment variables in appendix B.

4.14 XML Section: Others

```
<others enabled="yes">
  <minimum-time enabled="no">10m</minimum-time>
</others>
```

This section contains other configuration details that do not fit in the previous sections. Right now, there is only one option available and it is devoted to tell the instrumentation package the minimum instrumentation time. To enable it, set `enabled` to "yes" and set the minimum time within the `minimum-time` tag.

4.15 XML Section: Sampling

```
<sampling enabled="no" type="default" period="50m" />
```

This sections configures the time-based sampling capabilities. Every sample contains processor performance counters (if enabled in section 4.9.1 and either PAPI or PMAPI are referred at configure time) and callstack information (if enabled in section 4.7 and proper dependencies are set at configure time).

This section contains two attributes besides `enabled`. These are

- **type**: determines which timer domain is used (see `man 2 setitimer` or `man 3p setitimer` for further information on time domains). Available options are: `real` (which is also the `default` value, `virtual` and `prof`.
- **period**: specifies the sampling periodicity. In the example above, samples are gathered every 50ms.

See **EXTRAE_SAMPLING_PERIOD**, **EXTRAE_SAMPLING_CLOCKTYPE** and **EXTRAE_SAMPLING_CALLER** environment variables in appendix B.

4.16 XML Section: Merge

```
<merge enabled="yes"
  synchronization="default"
  binary="mpi_ping"
  tree-fan-out="16"
  max-memory="512"
  joint-states="yes"
  keep-mpits="yes"
  sort-addresses="yes"
>
  mpi_ping.prv
</merge>
```

If this section is enabled and the instrumentation packaged is configured to support this, the merge process will be automatically invoked after the application run. The merge process will use all the resources devoted to run the application.

The leaf of this node will be used as the tracefile name (`mpi_ping.prv` in this example). Current available options for the merge process are given as attribute of the `<merge>` node and they are:

- `synchronization`: which can be set to `default`, `node`, `task`, `no`. This determines how task clocks will be synchronized (*default is node*).
- `binary`: points to the binary that is being executed. It will be used to translate gathered addresses (MPI callers, sampling points and user functions) into source code references.
- `tree-fan-out`: *only for MPI executions* sets the tree-based topology to run the merger in a parallel fashion.
- `max-memory`: limits the intermediate merging process to run up to the specified limit (in MBytes).
- `joint-states`: which can be set to `yes`, `no`. Determines if the resulting Paraver tracefile will split or join equal consecutive states (*default is yes*).
- `keep-mpits`: whether to keep the intermediate tracefiles after performing the merge (*currently unimplemented*).
- `sort-addresses`: whether to sort all addresses that refer to the source code (enabled by default).

For further references, see chapter 6.

4.17 Using environment variables within the XML file

XML tags and attributes can refer to environment variables that are defined in the environment during the application run. If you want to refer to an environment variable within the XML file, just enclose the name of the variable using the dollar symbol (`$`), for example: `$FOO$`.

Note that the user has to put an specific value or a reference to an environment variable which means that expanding environment variables in text is not allowed as in a regular shell (i.e., the instrumentation package will not convert the follwing text `bar$FOO$bar`).

Chapter 5

Extrae API

There are two levels of the API in the Extrae instrumentation package. Basic API refers to the basic functionality provided and includes emitting events, source code tracking, changing instrumentation mode and so. Extended API is an *experimental* addition to provide several of the basic API within single and powerful calls using specific data structures.

5.1 Basic API

The following routines are defined in the `${EXTRAE_HOME}/include/extrae_user_events.h`. These routines are intended to be called by C/C++ programs. The instrumentation package also provides bindings for Fortran applications. The Fortran API bindings have the same name as the C API but honoring the Fortran compiler function name mangling scheme. The Extrae constants for Fortran applications can be included from `${EXTRAE_HOME}/include/extraef_user_events.h`.

- `void Extrae_init (void)`
  Initializes the tracing library.
  NOTE: This routine is called automatically by either `MPI_Init` or when tracing is performed by the DynInst launcher.

- `void Extrae_fini (void)`
  Finalizes the tracing library and dumps the intermediate tracing buffers onto disk.
  NOTE: This routine is called automatically by either `MPI_Finalize` or when the tracing is performed by the DynInst launcher.

- `void Extrae_event (unsigned event, unsigned value)`
  The Extrae_event adds a single timestamped event into the tracefile. The event has two arguments: type and value.

  Some common use of events are:

  – Identify loop iterations (or any code block): Given a loop, the user can set a unique type for the loop and a value related to the iterator value of the loop. For example:

```
for (i = 1; i <= MAX_ITERS; i++)
{
  Extrae_event (1000, i);
```

```
  [original loop code]
}
Extrae_event (1000, 0);
```

The last added call to Extrae_event marks the end of the loop setting the event value to 0, which facilitates the analysis with Paraver.

– Identify user routines: Choosing a constant type (6000019 in this example) and different values for different routines (set to 0 to mark a "leave" event)

```
void routine1 (void)
{
 Extrae_event (6000019, 1);
 [routine 1 code]
 Extrae_event (6000019, 0);
}

void routine2 (void)
{
 Extrae_event (6000019, 2);
 [routine 2 code]
 Extrae_event (6000019, 0);
}
```

– Identify any point in the application using a unique combination of type and value.

- `void Extrae_nevent (unsigned count, unsigned *types, unsigned *values)`
  Allows the user to place *count* events with the same timestamp at the given position.
- `void Extrae_counters (void)`
  Emits the value of the active hardware counters set. See chapter 4 for further information.
- `void Extrae_eventandcounters (int event, int value)`
  This routine lets the user add an event and obtain the performance counters with one call and a single timestamp.
- `void Extrae_neventandcounters (unsigned count, unsigned *types, unsigned *values)`
  This routine lets the user add several events and obtain the performance counters with one call and a single timestamp.
- `void Extrae_shutdown (void)`
  Turns off the instrumentation.
- `void Extrae_restart (void)`
  Turns on the instrumentation.
- `void Extrae_previous_hwc_set (void)`
  Makes the previous hardware counter set defined in the XML file to be the active set (see section 4.2 for further information).

- `void Extrae_next_hwc_set (void)`
  Makes the following hardware counter set defined in the XML file to be the active set (see section 4.2 for further information).

- `void Extrae_set_tracing_tasks (int from, int to)`
  Allows the user to choose from which tasks (not *threads*!) store informartion in the tracefile

- `void Extrae_set_options (int options)`
  Permits configuring several tracing options at runtime. The `options` parameter has to be a bitwise or combination of the following options, depending on the user's needs:

  - `EXTRAE_CALLER_OPTION`
    Dumps caller information at each entry or exit point of the MPI routines. Caller levels need to be configured at XML (see chapter 4).
  - `EXTRAE_HWC_OPTION`
    Activates hardware counter gathering.
  - `EXTRAE_MPI_OPTION`
    Activates tracing of MPI calls.
  - `EXTRAE_MPI_HWC_OPTION`
    Activates hardware counter gathering in MPI routines.
  - `EXTRAE_OMP_OPTION`
    Activates tracing of OpenMP runtime or outlined routines.
  - `EXTRAE_OMP_HWC_OPTION`
    Activates hardware counter gathering in OpenMP runtime or outlined routines.
  - `EXTRAE_UF_HWC_OPTION`
    Activates hardware counter gathering in the user functions.

- `void Extrae_network_counters (void)`
  Emits the value of the network counters if the system has this capability. *(Only available for systems with Myrinet GM/MX networks).*

- `void Extrae_network_routes (int task)`
  Emits the network routes for an specific `task`. *(Only available for systems with Myrinet GM/MX networks).*

- `void Extrae_user_function (int enter)`
  Emits an event into the tracefile which references the source code (data includes: source line number, file name and function name). If `enter` is 0 it marks an end (i.e., leaving the function), otherwise it marks the beginning of the routine. The user must be careful to place the call of this routine in places where the code is always executed, being careful not to place them inside `if` and `return` statements.

```
void routine1 (void)
{
 Extrae_user_function (1);
 [routine 1 code]
```

```
 Extrae_user_function (0);
}

void routine2 (void)
{
 Extrae_user_function (1);
 [routine 2 code]
 Extrae_user_function (0);
}
```

In order to gather performance counters during the execution of these calls, the `user-functions` tag in the XML configuration and its `counters` have to be both enabled.

5.2 Extended API

NOTE: *This API is in experimental stage. Use it at your own risk!*

The extended API makes use of two special structures located in `${PREFIX}/include/extrae_types.h`. The structures are `extrae_UserCommunication` and `extrae_CombinedEvents`. The former is intended to encode an event that will be converted into a Paraver communication when its partner equivalent event has found. The latter is used to generate events containing multiple kinds of information at the same time.

```
struct extrae_UserCommunication
{
  extrae_user_communication_types_t type;
  extrae_comm_tag_t tag;
  unsigned size; /* size_t? */
  extrae_comm_partner_t partner;
  extrae_comm_id_t id;
};
```

The structure `extrae_UserCommunication` contains the following fields:

- `type`
  Available options are:
  - `EXTRAE_USER_SEND`, if this event represents a send point.
  - `EXTRAE_USER_RECV`, if this event represents a receive point.
- `tag`
  The tag information in the communication record.
- `size`
  The size information in the communication record.

- `partner`
  The partner of this communication (receive if this is a send or send if this is a receive). Partners (ranging from 0 to N-1) are considered across tasks whereas all threads share a single communication queue.

- `id`
  An identifier that is used to match communications between partners.

```
struct extrae_CombinedEvents
{
  /* These are used as boolean values */
  int HardwareCounters;
  int Callers;
  int UserFunction;
  /* These are intended for N events */
  unsigned nEvents;
  extrae_type_t  *Types;
  extrae_value_t *Values;
  /* These are intended for user communication records */
  unsigned nCommunications;
  extrae_user_communication_t *Communications;
};
```

The structure `extrae_CombinedEvents` contains the following fields:

- `HardwareCounters`
  Set to non-zero if this event has to gather hardware performance counters.

- `Callers`
  Set to non-zero if this event has to emit callstack information.

- `UserFunction`
  Available options are:

  - `EXTRAE_USER_FUNCTION_NONE`, if this event should not provide information about user routines.
  - `EXTRAE_USER_FUNCTION_ENTER`, if this event represents the starting point of a user routine.
  - `EXTRAE_USER_FUNCTION_LEAVE`, if this event represents the ending point of a user routine.

- `nEvents`
  Set the number of events given in the `Types` and `Values` fields.

- `Types`
  A pointer containing `nEvents` type that will be stored in the trace.

- `Values`
  A pointer containing `nEvents` values that will be stored in the trace.

- `nCommunications`
  Set the number of communications given in the `Communications` field.

- `Communications`
  A pointer to `extrae_UserCommunication` structures containing `nCommunications` elements that represent the involved communications.

The extended API contains the following routines:

- `void Extrae_init_UserCommunication (struct extrae_UserCommunication *);`
  Use this routine to initialize an extrae_UserCommunication structure.

- `void Extrae_init_CombinedEvents (struct extrae_CombinedEvents *);`
  Use this routine to initialize an extrae_CombinedEvents structure.

- `void Extrae_emit_CombinedEvents (struct extrae_CombinedEvents *);`
  Use this routine to emit to the tracefile the events set in the extrae_CombinedEvents given.

- `void Extrae_resume_virtual_thread (unsigned vthread);`
  This routine changes the thread identifier so as to be vthread in the final tracefile. *Improper use of this routine may result in corrupt tracefiles.*

- `void Extrae_suspend_virtual_thread (void);`
  This routine recovers the original thread identifier (given by routines like pthread_self or omp_get_thread_num, for instance).

- `void Extrae_set_threadid_function (unsigned (*threadid_function)(void));`
  Defines the routine that will be used as a thread identifier inside the tracing facility.

- `void Extrae_set_numthreads_function (unsigned (*numthreads_function)(void));`
  Defines the routine that will count all the executing threads inside the tracing facility.

- `void Extrae_set_taskid_function (unsigned (*taskid_function)(void));`
  Defines the routine that will be used as a task identifier inside the tracing facility.

- `void Extrae_set_numtasks_function (unsigned (*numtasks_function)(void));`
  Defines the routine that will count all the executing tasks inside the tracing facility.

- `void Extrae_set_barrier_tasks_function (void (*barriertasks_function)(void));`
  Establishes the barrier routine among tasks. It is needed for synchronization purposes.

5.3 Special considerations for Cell Broadband Engine tracing package

Instead of including `${EXTRAE_HOME}/include/extrae_user_events.h` include:

- `${EXTRAE_HOME}/include/ppu_trace_sdk2.h` on the PPE side, and,

- `${EXTRAE_HOME}/include/sputrace_user_events.h` on the SPE side.

5.3.1 PPE side

The routines shown on section 5.1 are available for the PPE element. In addition, two additional routines are available to control the creation and finalization of the SPE threads. These routines are:

- `int CELLtrace_init (int spus, spe_context_ptr_t * spe_ids)`
  Contacts with the SPE thread to initialize once the SPE tracing environment. Such call has to be synchronized with the invocation of `SPUtrace_init` (see 5.3.2) call on the SPE side due to the presence of message passing using the mailboxes. The routine receives the total number of contexts created by the Cell SDK `spe_context_create` and a vector pointing to those contexts. Each of those contexts will reference to a single SPE thread created by a call to `pthread_create`.

- `void CELLtrace_fini (void)`
  Waits for the finalization of all the threads registered in `CELLtrace_init` and dumps their intermediate tracing buffers.

5.3.2 SPE side

Due to the lack of parallel paradigms and hardware counters inside the SPE element, the SPE tracing library is a subset of the typical tracing library. The following API calls are available for the SPE element:

- `void SPUtrace_init (void)`
  Initializes the tracing package in the SPE side. It has to be synchronized with `CELLtrace_init` due to the message passing using mailboxes.

- `void SPUtrace_fini (void)`
  Notifies the finalization of the work performed in the SPE thread and transfers the tracing buffer to the PPE element.

- `void SPUtrace_event (unsigned event, unsigned value)`
  Has the same semantics as `Extrae_event`.

- `void SPUtrace_nevent (unsigned count, unsigned *types, unsigned *values)`
  Has the same semantics as `Extrae_nevent`.

Chapter 6

Merging process

Once the application has finished, and if the automatic merge process is not setup, the merge must be executed manually. Here we detail how to run the merge process manually.

The inserted probes in the instrumented binary are responsible for gathering performance metrics of each task/thread and for each of them several files are created where the XML configuration file specified (see section 4.10). Such files are:

- As many `.mpit` files as tasks and threads where running the target application. Each file contains information gathered by the specified task/thread in raw binary format.
- A single `.mpits` file that contain a list of related `.mpit` files.
- If the DynInst based instrumentation package was used, an addition `.sym` file that contains some symbolic information gathered by the DynInst library.

In order to use Paraver, those intermediate files (*i.e.*, `.mpit` files) must be merged and translated into Paraver trace file format. The same applies if the user wants to use the Dimemas simulator. To proceed with any of these translation all the intermediate trace files must be merged into a single trace file using one of the available mergers in the `bin` directory (see table 6.1).

The target trace type is defined in the XML configuration file used at the instrumentation step (see section 4.1), and it has match with the merger used (mpi2prv and mpimpi2prv for Paraver and mpi2dim and mpimpi2dim for Dimemas). However, it is possible to force the format nevertheless the selection done in the XML file using the parameters `-paraver` or `-dimemas`[1].

Binary	**Description**
mpi2prv	Sequential version of the Paraver merger.
mpi2dim	Sequential version of the Dimemas merger.
mpimpi2prv	Parallel version of the Paraver merger.
mpimpi2dim	Parallel version of the Dimemas merger.

Table 6.1: Description of the available mergers in the Extrae package.

[1]The timing mechanism differ in Paraver/Dimemas at the instrumentation level. If the output trace format does not correspond with that selected in the XML some timing inaccuracies may be present in the final tracefile. Such inaccuracies are known to be higher due to clock granularity if the XML is set to obtain Dimemas tracefiles but the resulting tracefile is forced to be in Paraver format.

6.1 Paraver merger

As stated before, there are two Paraver mergers: `mpi2prv` and `mpimpi2prv`. The former is for use in a single processor mode while the latter is meant to be used with multiple processors using MPI (and cannot be run using one MPI task).

Paraver merger receives a set of intermediate trace files and generates three files with the same name (which is set with the `-o` option) but differ in the extension. The Paraver trace itself (.prv file) that contains timestamped records that represent the information gathered during the execution of the instrumented application. It also generates the Paraver Configuration File (.pcf file), which is responsible for translating values contained in the Paraver trace into a more human readable values. Finally, it also generates a file containing the distribution of the application across the cluster computation resources (.row file).

The following sections describe the available options for the Paraver mergers. Typically, options available for single processor mode are also available in the parallel version, unless specified.

6.1.1 Sequential Paraver merger

These are the available options for the sequential Paraver merger:

- `-d` Dumps the information stored in the intermediate trace files.
- `-e BINARY` Uses the given BINARY to translate addresses that are stored in the intermediate trace files into useful information (including function name, source file and line). The application has to be compiled with `-g` flag so as to obtain valuable information.
- `-evtnum N`
  Partially processes (up to N events) the intermediate trace files to generate the Dimemas tracefile.
- `-f FILE.mpits` *(where* `FILE.mpits` *file is generated by the instrumentation)*
  The merger uses the given file (which contains a list of intermediate trace files of a single executions) instead of giving set of intermediate trace files.
  This option looks first for each file listed in the parameter file. Each contained file is searched in the absolute given path, if it does not exist, then it's searched in the current directory.
- `-f-relative FILE.mpits` *(where* `FILE.mpits` *file is generated by the instrumentation)*
  This options behaves like the -f options but looks for the intermediate files in the current directory.
- `-f-absolute FILE.mpits` *(where* `FILE.mpits` *file is generated by the instrumentation)*
  This options behaves like the -f options but uses the full path of every intermediate file so as to locate them.
- `-h`
  Provides minimal help about merger options.
- `-maxmem M`
  The last step of the merging process will be limited to use M megabytes of memory. By default, M is 512.

- `-s FILE.sym` *(where* `FILE.sym` *file is generated with the DynInst instrumentator)*
  Passes information regarding instrumented symbols into the merger to aid the Paraver analysis. If `-f`, `-f-relative` or `-f-absolute` paramters are given, the merge process will try to automatically load the symbol file associated to that FILE.mpits file.

- `-no-syn`
  If set, the merger will not attempt to synchronize the different tasks. This is useful when merging intermediate files obtained from a single node (and thus, share a single clock).

- `-o FILE.prv`
  Choose the name of the target Paraver tracefile.

- `-o FILE.prv.gz`
  Choose the name of the target Paraver tracefile compressed using the libz library.

- `-skip-sendrecv`
  Do not match point to point communications issued by `MPI_Sendrecv` or `MPI_Sendrecv_replace`.

- `-sort-addresses`
  Sort event values that reference source code locations so as the values are sorted by file name first and then line number (enabled by default).

- `-split-states`
  Do not join consecutive states that are the same into a single one.

- `-syn`
  If different nodes are used in the execution of a tracing run, there can exist some clock differences on all the nodes. This option makes mpi2prv to recalculate all the timings based on the end of the MPI_Init call. This will usually lead to "synchronized" tasks, but it will depend on how the clocks advance in time.

- `-syn-node`
  If different nodes are used in the execution of a tracing run, there can exist some clock differences on all the nodes. This option makes mpi2prv to recalculate all the timings based on the end of the MPI_Init call and the node where they ran. This will usually lead to better synchronized tasks than using -syn, but, again, it will depend on how the clocks advance in time.

- `-unique-caller-id`
  Choose whether use a unique value identifier for different callers locations (MPI calling routines, user routines, OpenMP outlined routines andpthread routines).

6.1.2 Parallel Paraver merger

These options are specific to the parallel version of the Paraver merger:

- `-block`
  Intermediate trace files will be distributed in a block fashion instead of a cyclic fashion to the merger.

- `-cyclic`
  Intermediate trace files will be distributed in a cyclic fashion instead of a block fashion to the merger.

- `-size`
  The intermediate trace files will be sorted by size and then assigned to processors in a such manner that each processor receives approximately the same size.

- `-consecutive-size`
  Intermediate trace files will be distributed consecutively to processors but trying to distribute the overall size equally among processors.

- `-use-disk-for-comms`
  Use this option if your memory resources are limited. This option uses an alternative matching communication algorithm that saves memory but uses intensively the disk.

- `-tree-fan-out N`
  Use this option to instruct the merger to generate the tracefile using a tree-based topology. This should improve the performance when using a large number of processes at the merge step. Depending on the combination of processes and the width of the tree, the merger will need to run several stages to generate the final tracefile.
  The number of processes used in the merge process must be equal or greater than the N parameter. If it is not, the merger itself will automatically set the width of the tree to the number of processes used.

6.2 Dimemas merger

As stated before, there are two Dimemas mergers: `mpi2dim` and `mpimpi2dim`. The former is for use in a single processor mode while the latter is meant to be used with multiple processors using MPI.

In contrast with Paraver merger, Dimemas mergers generate a single output file with the .dim extension that is suitable for the Dimemas simulator from the given intermediate trace files..

These are the available options for both Dimemas mergers:

- `-evtnum N`
  Partially processes (up to N events) the intermediate trace files to generate the Dimemas tracefile.

- `-f FILE.mpits` *(where* `FILE.mpits` *file is generated by the instrumentation)*
  The merger uses the given file (which contains a list of intermediate trace files of a single executions) instead of giving set of intermediate trace files.
  This option takes only the file name of every intermediate file so as to locate them.

- `-f-relative FILE.mpits` *(where* `FILE.mpits` *file is generated by the instrumentation)*
  This options works exactly as the -f option.

- `-f-absolute FILE.mpits` *(where* `FILE.mpits` *file is generated by the instrumentation)*
  This options behaves like the -f options but uses the full path of every intermediate file so as to locate them.

- `-h`
  Provides minimal help about merger options.

- `-maxmem M`
  The last step of the merging process will be limited to use M megabytes of memory. By default, M is 512.

- `-o FILE.dim`
  Choose the name of the target Dimemas tracefile.

6.3 Environment variables

There are some environment variables that are related Two environment variables

6.3.1 Environment variables suitable to Paraver merger

EXTRAE_LABELS

This environment variable lets the user add custom information to the generated Paraver Configuration File (`.pcf`). Just set this variable to point to a file containing labels for the unknown (user) events.

The format for the file is:

```
EVENT_TYPE
0 [type1] [label1]
0 [type2] [label2]
...
0 [typeK] [labelK]
```

Where `[typeN]` is the event value and `[labelN]` is the description for the event with value [typeN]. It is also possible to link both type and value of an event:

```
EVENT_TYPE
0 [type] [label]
VALUES
[value1] [label1]
[value2] [label2]
...
[valueN] [labelN]
```

With this information, Paraver can deal with both type and value when giving textual information to the end user. If Paraver does not find any information for an event/type it will shown it in numerical form.

MPI2PRV_TMP_DIR

Points to a directory where all intermediate temporary files will be stored. These files will be removed as soon the application ends.

6.3.2 Environment variables suitable to Dimemas merger

MPI2DIM_TMP_DIR

Points to a directory where all intermediate temporary files will be stored. These files will be removed as soon the application ends.

Chapter 7

Examples

We present here three different examples of generating a Paraver tracefile. First example requires the package to be compiled with DynInst libraries. Second example uses the `LD_PRELOAD` or `LDR_PRELOAD[64]` mechanism to interpose code in the application. Such mechanism is available in Linux and FreeBSD operating systems and only works when the application uses dynamic libraries. Finally, there is an example using the static library of the instrumentation package.

7.1 DynInst based examples

DynInst is a third-party instrumentation library developed at UW Madison which can instrument in-memory binaries. It adds flexibility to add instrumentation to the application without modifying the source code. DynInst is ported to different systems (Linux, FreeBSD) and to different architectures[1] (x86, x86/64, PPC32, PPC64) but the functionality is common to all of them.

7.1.1 Generating intermediate files for serial or OpenMP applications

run_dyninst.sh

```
1 #!/bin/sh
2
3 export EXTRAE_HOME=WRITE-HERE-THE-PACKAGE-LOCATION
4 export LD_LIBRARY_PATH=${EXTRAE_HOME}/lib
5 source ${EXTRAE_HOME}/etc/extrae.sh
6
7 ## Run the desired program
8 ${EXTRAE_HOME}/bin/extrae -config extrae.xml $*
```

A similar script can be found in the share/example/SEQ directory in your tracing package directory. Just tune the `EXTRAE_HOME` environment variable and make the script executable (using `chmod u+x`). You can either pass the XML configuration file through the `EXTRAE_CONFIG_FILE` if you prefer instead. Line no. 5 is responsible for loading all the environment variables needed for the DynInst launcher (called extrae) that is invoked in line 8.

In fact, there are two examples provided in share/example/SEQ, one for static (or manual) instrumentation and another for the DynInst-based instrumentation. When using the DynInst

[1] The IA-64 architecture support was dropped by DynInst 7.0

instrumentation, the user may add new routines to instrument using the existing `function-list` file that is already pointed by the `extrae.xml` configuration file. The way to specify the routines to instrument is add as many lines with the name of every routine to be instrumented.

Running OpenMP applications using DynInst is rather similar to serial codes. Just compile the application with the appropiate OpenMP flags and run as before. You can find an example in the share/example/OMP directory.

7.1.2 Generating intermediate files for MPI applications

MPI applications can also be instrumented using the DynInst instrumentator. The instrumentation is done independently to each spawned MPI process, so in order to execute the DynInst-based instrumentation package on a MPI application, you must be sure that your MPI launcher supports running shell-scripts. The following scripts show how to run the DynInst instrumentator from the MOAB/Slurm queue system. The first script just sets the environment for the job whereas the second is responsible for instrumenting every spawned task.

slurm_trace.sh

```
1  #!/bin/bash
2  # @ initialdir = .
3  # @ output = trace.out
4  # @ error =  trace.err
5  # @ total_tasks = 4
6  # @ cpus_per_task = 1
7  # @ tasks_per_node = 4
8  # @ wall_clock_limit = 00:10:00
9  # @ tracing = 1
10
11 srun ./run.sh ./mpi_ping
```

The most important thing in the previous script is the line number 11, which is responsible for spawning the MPI tasks (using the srun command). The spawn method is told to execute `./run.sh ./mpi_ping` which in fact refers to instrument the `mpi_ping` binary using the `run.sh` script. You must adapt this file to your queue-system (if any) and to your MPI submission mechanism (i.e., change srun to mpirun, mpiexec, poe, *etc*...). Note that changing the line 11 to read like `./run.sh srun ./mpi_ping` would result in instrumenting the srun application not mpi_ping.

run.sh

```
1  #!/bin/bash
2
3  export EXTRAE_HOME=@sub_PREFIXDIR@
4  source ${EXTRAE_HOME}/etc/extrae.sh
5
6  # Only show output for task 0, others task send output to /dev/null
7  if test "${SLURM_PROCID}" == "0" ; then
8    ${EXTRAE_HOME}/bin/extrae -config ../extrae.xml $@ > job.out 2> job.err
9  else
10   ${EXTRAE_HOME}/bin/extrae -config ../extrae.xml $@ > /dev/null 2> /dev/null
11 fi
```

This is the script responsible for instrumenting a single MPI task. In line number 4 we set-up the instrumentation environment by executing the commands from `extrae.sh`. Then we execute the binary passed to the `run.sh` script in lines 8 and 10. Both lines are executing the same command except that line 8 sends all the output to two different files (one for standard output and another for standard error) and line 10 sends all the output to `/dev/null`.

Please note, this script is particularly adapted to the MOAB/Slurm queue systems. You may need to adapt the script to other systems by using the appropiate environment variables. Particularly, `SLURM_PROCID` identifies the MPI task id (i.e., the task rank) and may be changed to the proper environemnt variable (PMI_RANK in ParaStation/Torque/MOAB system or MXMPI_ID in systems having Myrinet MX devices, for example).

7.2 LD_PRELOAD based examples

LD_PRELOAD (or LDR_PRELOAD[64] in AIX) interposition mechanism only works for binaries that are linked against shared libraries. This interposition is done by the runtime loader by substituting the original symbols by those provided by the instrumentation package. This mechanism is known to work on Linux, FreeBSD and AIX operating systems, although it may be available on other operating systems (even using different names[2]) they are not tested.

We show how this mechanism works on Linux (or similar environments) in subsection 7.2.1 and on AIX in subsection 7.2.3.

7.2.1 Linux

The following script preloads the libmpitrace library to instrument MPI calls of the application passed as an argument (tune `EXTRAE_HOME` according to your installation).

trace.sh

```
1 #!/bin/sh
2
3 export EXTRAE_HOME=WRITE-HERE-THE-PACKAGE-LOCATION
4 export EXTRAE_CONFIG_FILE=extrae.xml
5 export LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitrace.so
6
7 ## Run the desired program
8 $*
```

The previous script can be found in the share/example/MPI/ld-preload directory in your tracing package directory. Copy the script to one of your directories, tune the `EXTRAE_HOME` environment variable and make the script executable (using `chmod u+x`). Also copy the XML configuration extrae.xml file from the share/example/MPI directory instrumentation package to the current directory. This file is used to configure the whole behavior of the instrumentation package (there is more information about the XML file on chapter 4). The last line in the script, $*, executes the arguments given to the script, so as you can run the instrumentation by simply adding the script in between your execution command.

[2]Look at `http://www.fortran-2000.com/ArnaudRecipes/sharedlib.html` for further information.

Regarding the execution, if you run MPI applications from the command-line, you can issue the typical mpirun command as:

```
${MPI_HOME}/bin/mpirun -np N ./trace.sh mpi-app
```

where, `${MPI_HOME}` is the directory for your MPI installation, `N` is the number of MPI tasks you want to run and `mpi-app` is the binary of the MPI application you want to run.

However, if you execute your MPI applications through a queue system you may need to write a submission script. The following script is an example of a submission script for MOAB/Slurm queuing system using the aforementioned trace.sh script for an execution of the `mpi-app` on two processors.

slurm-trace.sh

```
1  #! /bin/bash
2  #@ job_name         = trace_run
3  #@ output           = trace_run%j.out
4  #@ error            = trace_run%j.out
5  #@ initialdir       = .
6  #@ class            = bsc_cs
7  #@ total_tasks      = 2
8  #@ wall_clock_limit = 00:30:00
9
10 srun ./trace.sh mpi_app
```

If your system uses LoadLeveler your job script may look like:

ll.sh

```
1  #! /bin/bash
2  #@ job_type = parallel
3  #@ output = trace_run.ouput
4  #@ error = trace_run.error
5  #@ blocking = unlimited
6  #@ total_tasks = 2
7  #@ class = debug
8  #@ wall_clock_limit = 00:10:00
9  #@ restart = no
10 #@ group = bsc41
11 #@ queue
12
13 export MLIST=/tmp/machine_list ${$}
14 /opt/ibmll/LoadL/full/bin/ll_get_machine_list > ${MLIST}
15 set NP = ‘cat ${MLIST} | wc -l‘
16
17 ${MPI_HOME}/mpirun -np ${NP} -machinefile ${MLIST} ./trace.sh ./mpi-app
18
19 rm ${MLIST}
```

Besides the job specification given in lines 1-11, there are commands of particular interest. Lines 13-15 are used to know which and how many nodes are involved in the computation. Such

information information is given to the `mpirun` command to proceed with the execution. Once the execution finished, the temporal file created on line 14 is removed on line 19.

7.2.2 CUDA

There are two ways to instrument CUDA applications, depending on how the package was configured. If the package was configure with `--enable-cuda` only interposition on binaries using shared libraries are available. If the package was configured with `--with-cupti` any kind of binary can be instrumented because the instrumentation relies on the CUPTI library to instrument CUDA calls. The example shown below is intended for the former case.

run.sh

```
1 #!/bin/bash
2
3 export EXTRAE_HOME=/home/harald/extrae
4 export PAPI_HOME=/home/harald/aplic/papi/4.1.4
5
6 EXTRAE_CONFIG_FILE=extrae.xml LD_LIBRARY_PATH=${EXTRAE_HOME}/lib:${PAPI_HOME}/lib:${LD_LIBR
7 ${EXTRAE_HOME}/bin/mpi2prv -f TRACE.mpits -e ./hello
```

In this example, the hello application is compiled using the `nvcc` compiler and linked against the `-lcudatrace` library. The binary contains calls to `Extrae_init` and `Extrae_fini` and then executes a CUDA kernel. Line number 6 refers to the execution of the application itself. The Extrae configuration file and the location of the shared libraries are set in this line. Line number 7 invokes the merge process to generate the final tracefile.

7.2.3 AIX

AIX typically ships with POE and LoadLeveler as MPI implementation and queue system respectively. An example for a system with these software packages is given below. Please, note that the example is intended for 64 bit applications, if using 32 bit applications then `LDR_PRELOAD64` needs to be changed in favour of `LDR_PRELOAD`.

ll-aix64.sh

```
1  #@ job_name = basic_test
2  #@ output = basic_stdout
3  #@ error = basic_stderr
4  #@ shell = /bin/bash
5  #@ job_type = parallel
6  #@ total_tasks = 8
7  #@ wall_clock_limit = 00:15:00
8  #@ queue
9
10 export EXTRAE_HOME=WRITE-HERE-THE-PACKAGE-LOCATION
11 export EXTRAE_CONFIG_FILE=extrae.xml
12 export LDR_PRELOAD64=${EXTRAE_HOME}/lib/libmpitrace.so
13
14 ./mpi-app
```

Lines 1-8 contain a basic LoadLeveler job definition. Line 10 sets the Extrae package directory in EXTRAE_HOME environment variable. Follows setting the XML configuration file that will be used to set up the tracing. Then follows setting `LDR_PRELOAD64` which is responsible for instrumentation using the shared library `libmpitrace.so`. Finally, line 14 executes the application binary.

7.3 Statically linked based examples

This is the basic instrumentation method suited for those installations that neither support DynInst nor LD_PRELOAD, or require adding some manual calls to the Extrae API.

7.3.1 Linking the application

To get the instrumentation working on your code, first you have to link your application with the Extrae libraries. There are installed examples in your package distribution under share/examples directory. There you can find MPI, OpenMP, pthread and sequential examples depending on the support at configure time.

Consider the example Makefile found in share/examples/MPI/static:

Makefile

```
1  MPI_HOME = /gpfs/apps/MPICH2/mx/1.0.7..2/64
2  EXTRAE_HOME = /home/bsc41/bsc41273/foreign-pkgs/extrae-11oct-mpich2/64
3  PAPI_HOME = /gpfs/apps/PAPI/3.6.2-970mp-patched/64
4  XML2_LDFLAGS = -L/usr/lib64
5  XML2_LIBS = -lxml2
6
7  F77 = $(MPI_HOME)/bin/mpif77
8  FFLAGS = -O2
9  FLIBS = $(EXTRAE_HOME)/lib/libmpitracef.a \
10         -L$(PAPI_HOME)/lib -lpapi -lperfctr \
11         $(XML2_LDFLAGS) $(XML2_LIBS)
12
13 all: mpi_ping
14
15 mpi_ping: mpi_ping.f
16         $(F77) $(FFLAGS) mpi_ping.f $(FLIBS) -o mpi_ping
17
18 clean:
19         rm -f mpi_ping *.o pingtmp? TRACE.*
```

Lines 2-5 are definitions of some Makefile variables to set up the location of different packages needed by the instrumentation. In particular, `EXTRAE_HOME` sets where the Extrae package directory is located. In order to link your application with Extrae you have to add its libraries in the link stage (see lines 9-11 and 16). Besides `libmpitracef.a` we also add some PAPI libraries (`-lpapi`, and its dependency (which you may or not need `-lperfctr`), the libxml2 parsing library (`-lxml2`), and finally, the bfd and liberty libraries (`-lbfd` and `-liberty`), if the instrumentation package was compiled to support merge after trace (see chapter 3 for further information).

7.3.2 Generating the intermediate files

Executing an application with the statically linked version of the instrumentation package is very similar as the method shown in Section 7.2. There is, however, a difference: do not set LD_PRELOAD in `trace.sh`.

trace.sh

```
#!/bin/sh

export EXTRAE_HOME=WRITE-HERE-THE-PACKAGE-LOCATION
export EXTRAE_CONFIG_FILE=extrae.xml
export LD_LIBRARY_PATH=${EXTRAE_HOME}/lib:\
                       /gpfs/apps/MPICH2/mx/1.0.7..2/64/lib:\
                       /gpfs/apps/PAPI/3.6.2-970mp-patched/64/lib

## Run the desired program
$*
```

See section 7.2 to know how to run this script either through command line or queue systems.

7.4 Generating the final tracefile

Independently from the tracing method chosen, it is necessary to translate the intermediate tracefiles into a Paraver tracefile. The Paraver tracefile can be generated automatically (if the tracing package and the XML configuration file were set up accordingly, see chapters 3 and 4) or manually. In case of using the automatic merging process, it will use all the resources allocated for the application to perform the merge once the application ends.

To manually generate the final Paraver tracefile issue the following command:

```
${EXTRAE_HOME}/bin/mpi2prv -f TRACE.mpits -e mpi-app -o trace.prv
```

This command will convert the intermediate files generated in the previous step into a single Paraver tracefile. The `TRACE.mpits` is a file generated automatically by the instrumentation and contains a reference to all the intermediate files generated during the execution run. The `-e` parameter receives the application binary `mpi-app` in order to perform translations from addresses to source code. To use this feature, the binary must have been compiled with debugging information. Finally, the `-o` flag tells the merger how the Paraver tracefile will be named (trace.prv in this case).

Appendix A

An example of Extrae XML configuration file

```
<?xml version='1.0'?>

<trace enabled="yes"
 home="@sed_MYPREFIXDIR@"
 initial-mode="detail"
 type="paraver"
 xml-parser-id="@sed_XMLID@"
>
  <mpi enabled="yes">
    <counters enabled="yes" />
  </mpi>

  <pacx enabled="no">
    <counters enabled="yes" />
  </pacx>

  <pthread enabled="yes">
    <locks enabled="no" />
    <counters enabled="yes" />
  </pthread>

  <openmp enabled="yes">
    <locks enabled="no" />
    <counters enabled="yes" />
  </openmp>

  <callers enabled="yes">
    <mpi enabled="yes">1-3</mpi>
    <pacx enabled="no">1-3</pacx>
    <sampling enabled="no">1-5</sampling>
```

```
</callers>

<user-functions enabled="no" list="/home/bsc41/bsc41273/user-functions.dat">
  <counters enabled="yes" />
</user-functions>

<counters enabled="yes">
  <cpu enabled="yes" starting-set-distribution="1">
    <set enabled="yes" domain="all" changeat-globalops="5">
      PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM
      <sampling enabled="no" period="100000000">PAPI_TOT_CYC</sampling>
    </set>
    <set enabled="yes" domain="user" changeat-globalops="5">
      PAPI_TOT_INS,PAPI_FP_INS,PAPI_TOT_CYC
    </set>
  </cpu>
  <network enabled="yes" />
  <resource-usage enabled="yes" />
</counters>

<storage enabled="no">
  <trace-prefix enabled="yes">TRACE</trace-prefix>
  <size enabled="no">5</size>
  <temporal-directory enabled="yes">/scratch</temporal-directory>
  <final-directory enabled="yes">/gpfs/scratch/bsc41/bsc41273</final-directory>
  <gather-mpits enabled="no" />
</storage>

<buffer enabled="yes">
  <size enabled="yes">150000</size>
  <circular enabled="no" />
</buffer>

<trace-control enabled="yes">
  <file enabled="no" frequency="5M">/gpfs/scratch/bsc41/bsc41273/control</file>
  <global-ops enabled="no">10</global-ops>
  <remote-control enabled="yes">
    <mrnet enabled="yes" target="150" analysis="spectral" start-after="30">
      <clustering max_tasks="26" max_points="8000"/>
      <spectral min_seen="1" max_periods="0" num_iters="3" signals="DurBurst,InMPI"/>
    </mrnet>
    <signal enabled="no" which="USR1"/>
  </remote-control>
</trace-control>
```

```
  <others enabled="yes">
    <minimum-time enabled="no">10m</minimum-time>
  </others>

  <bursts enabled="no">
    <threshold enabled="yes">500u</threshold>
    <mpi-statistics enabled="yes" />
    <pacx-statistics enabled="no" />
  </bursts>

  <cell enabled="no">
    <spu-file-size enabled="yes">5</spu-file-size>
    <spu-buffer-size enabled="yes">64</spu-buffer-size>
    <spu-dma-channel enabled="yes">2</spu-dma-channel>
  </cell>

  <merge enabled="yes"
    synchronization="default"
    binary="mpi_ping"
    tree-fan-out="16"
    max-memory="512"
    joint-states="yes"
    keep-mpits="yes"
    sort-addresses="no"
  >
    mpi_ping.prv
  </merge>

</trace>
```

Appendix B

Environment variables

Although Extrae is configured through an XML file (which is pointed by the `EXTRAE_CONFIG_FILE`), it also supports minimal configuration to be done via environment variables for those systems that do not have the library responsible for parsing the XML files (*i.e.,* libxml2).

This appendix presents the environment variables the Extrae package uses if `EXTRAE_CONFIG_FILE` is not set and a description. For those environment variable that refer to XML 'enabled' attributes (*i.e.*, that can be set to "yes" or "no") are considered to be enabled if their value are defined to 1.

Environment variable	Description
EXTRAE_BUFFER_SIZE	Set the number of records that the instrumentation buffer can hold before flushing them.
EXTRAE_CIRCULAR_BUFFER	*(deprecated)*
EXTRAE_COUNTERS	See section 4.9.1. Just one set can be defined. Counters (in PAPI) groups (in PMAPI) are given separated by commas.
EXTRAE_CONTROL_FILE	The instrumentation will be enabled only when the file pointed exists.
EXTRAE_CONTROL_GLOPS	Starts the instrumentation when the specified number of global collectives have been executed.
EXTRAE_CONTROL_TIME	Checks the file pointed by `EXTRAE_CONTROL_FILE` at this period.
EXTRAE_DIR	Specifies where temporal files will be created during instrumentation.
EXTRAE_DISABLE_MPI	Disable MPI instrumentation.
EXTRAE_DISABLE_OMP	Disable OpenMP instrumentation.
EXTRAE_DISABLE_PTHREAD	Disable pthread instrumentation.
EXTRAE_DISABLE_PACX	Disable PACX instrumentation.
EXTRAE_FILE_SIZE	Set the maximum size (in Mbytes) for the intermediate trace file.
EXTRAE_FUNCTIONS	List of routine to be instrumented, as described in 4.8 using the GNU C `-finstrument-functions` or the IBM XL `-qdebug=function_trace` option at compile and link time.
EXTRAE_FUNCTIONS_COUNTERS_ON	Specify if the performance counters should be collected when a user function event is emitted.
EXTRAE_FINAL_DIR	Specifies where files will be stored when the application ends.
EXTRAE_GATHER_MPITS	Gather intermediate trace files into a single directory *(this is only available when instrumenting MPI applications).*
EXTRAE_HOME	Points where the Extrae is installed.
EXTRAE_INITIAL_MODE	Choose whether the instrumentation runs in in `detail` or in `bursts` mode.
EXTRAE_BURST_THRESHOLD	Specify the threshold time to filter running bursts.
EXTRAE_MINIMUM_TIME	Specify the minimum amount of instrumentation time.

Table B.1: Set of environment variables available to configure Extrae

Environment variable	Description
EXTRAE_MPI_CALLER	Choose which MPI calling routines should be dumped into the tracefile.
EXTRAE_MPI_COUNTERS_ON	Set to 1 if MPI must report performace counter values.
EXTRAE_MPI_STATISTICS	Set to 1 if basic MPI statistics must be collected in burst mode *(Only available in systems with Myrinet GM/MX networks).*
EXTRAE_NETWORK_COUNTERS	Set to 1 to dump network performance counters at flush points.
EXTRAE_PTHREAD_COUNTERS_ON	Set to 1 if pthread must report performance counters values.
EXTRAE_OMP_COUNTERS_ON	Set to 1 if OpenMP must report performance counters values.
EXTRAE_PTHREAD_LOCKS	Set to 1 if pthread locks have to be instrumented.
EXTRAE_OMP_LOCKS	Set to 1 if OpenMP locks have to be instrumented.
EXTRAE_ON	Enables instrumentation
EXTRAE_PACX_CALLER	Choose which PACX calling routines should be dumped into the tracefile.
EXTRAE_PACX_COUNTERS_ON	Set to 1 if PACX must report performace counter values.
EXTRAE_PACX_STATISTICS	Set to 1 if basic PACX statistics must be collected in burst mode.
EXTRAE_PROGRAM_NAME	Specify the prefix of the resulting intermediate trace files.
EXTRAE_SAMPLING_CALLER	Determines the callstack segment stored through time-sampling capabilities.
EXTRAE_SAMPLING_PERIOD	Enable time-sampling capabilities with the indicated period.
EXTRAE_SAMPLING_CLOCKTYPE	Determines domain for sampling clock. Options are: DEFAULT, REAL, VIRTUAL and PROF.
EXTRAE_RUSAGE	Instrumentation emits resource usage at flush points if set to 1.
EXTRAE_SPU_DMA_CHANNEL	Choose the SPU-PPU dma communication channel.
EXTRAE_SPU_BUFFER_SIZE	Set the buffer size of the SPU side.
EXTRAE_SPU_FILE_SIZE	Set the maximum size for the SPU side (default: 5Mbytes).
EXTRAE_TRACE_TYPE	Choose whether the resulting tracefiles are intended for Paraver or Dimemas.

Table B.2: Set of environment variables available to configure Extrae (*continued*)

Appendix C

Frequently Asked Questions

C.1 Configure, compile and link FAQ

- **Question:** The `bootstrap` script claims `libtool` errors like:
  ```
  src/common/Makefile.am:9:  Libtool library used but 'LIBTOOL' is undefined
  src/common/Makefile.am:9:  The usual way to define 'LIBTOOL' is to add 'AC_PROG_LIBTOOL
  src/common/Makefile.am:9:  to 'configure.ac' and run 'aclocal' and 'autoconf'
  again.
  src/common/Makefile.am:9:  If 'AC_PROG_LIBTOOL' is in 'configure.ac', make sure
  src/common/Makefile.am:9:  its definition is in aclocal's search path.
  ```
  Answer: Add to the `aclocal` (which is called in `bootstrap`) the directory where it can find the M4-macro files from `libtool`. Use the `-I option` to add it.

- **Question:** The `bootstrap` script claims that some macros are not found in the library, like:
  ```
  aclocal:configure.ac:338:  warning:  macro 'AM_PATH_XML2' not found in library
  ```
  Answer: Some M4 macros are not found. In this specific example, the libxml2 is not installed or cannot be found in the typical installation directory. To solve this issue, check whether the libxml2 is installed and modify the line in the `bootstrap` script that reads
  ```
  && aclocal -I config
  ```
  into
  ```
  && aclocal -I config -I/path/to/xml/m4/macros
  ```
  where `/path/to/xml/m4/macros` is the directory where the libxml2 M4 got installed (for example /usr/local/share/aclocal).

- **Question:** The application cannot be linked succesfully. The link stage complains about (or something similar like)
  ```
  ld:  0711-317 ERROR: Undefined symbol:  .__udivdi3.
  ld:  0711-317 ERROR: Undefined symbol:  .__mulvsi3.
  ```
  Answer: The instrumentation libraries have been compiled with GNU compilers whereas the application is compiled using IBM XL compilers. Add the libgcc_s library to the link stage of the application. This library can be found under the installation directory of the GNU compiler.

- **Question:** The make command dies when building libraries belonging Extrae in an AIX machine with messages like:

```
libtool: link: ar cru libcommon.a libcommon_la-utils.o libcommon_la-events.o
ar: 0707-126 libcommon_la-utils.o is not valid with the current object file mode.
Use the -X option to specify the desired object mode.
ar: 0707-126 libcommon_la-events.o is not valid with the current object file mode.
Use the -X option to specify the desired object mode.
```

Answer: `Libtool` uses `ar` command to build static libraries. However, `ar` does need special flags (-X64) to deal with 64 bit objects. To workaround this problem, just set the environment variable OBJECT_MODE to 64 before executing `gmake`. The `ar` command honors this variable to properly handle the object files in 64 bit mode.

- **Question:** The `configure` script dies saying
`configure: error: Unable to determine pthread library support`.
Answer: Some systems (like BG/L) does not provide a pthread library and `configure` claims that cannot find it. Launch the `configure` script with the `-disable-pthread` parameter.

- **Question:** NOT! `gmake` command fails when compiling the instrumentation package in a machine running AIX operating system, using 64 bit mode and IBM XL compilers complaining about Profile MPI (PMPI) symbols.
Answer: NOT! Use the reentrant version of IBM compilers (`xlc_r` and `xlC_r`). Non reentrant versions of MPI library does not include 64 bit MPI symbols, whereas reentrant versions do. To use these compilers, set the CC (C compiler) and CXX (C++ compiler) environment variables before running the `configure` script.

- **Question:** The compiler fails complaining that some parameters can not be understand when compiling the parallel merge. **Answer:** If the environment has more than one compiler (for example, IBM and GNU compilers), is it possible that the parallel merge compiler is not the same as the rest of the package. There are two ways to solve this:
  - Force the package compilation with the same backend as the parallel compiler. For example, for IBM compiler, set `CC=xlc` and `CXX=xlC` at the configure step.
  - Tell the parallel compiler to use the same compiler as the rest of the package. For example, for IBM compiler mpcc, set `MP_COMPILER=gcc` when issuing the make command.

- **Question:** The instrumentation package does not generate the shared instrumentation libraries but generates the satatic instrumentation libraries.
Answer 1: Check that the configure step was compiled without `--disable-shared` or force it to be enabled through `--enable-shared`.
Answer 2: Some MPI libraries (like MPICH 1.2.x) do not generate the shared libraries by default. The instrumentation package rely on them to generate its shared libraries, so make sure that the shared libraries of the MPI library are generated.

C.2 Execution FAQ

- **Question:** Why do the environment variables are not exported?
Answer: MPI applications are launched using special programs (like `mpirun, poe, mprun, srun...`) that spawn the application for the selected resources. Some of these programs do

not export all the environment variables to the spawned processes. Check if the the launching program does have special parameters to do that, or use the approach used on section 7 based on launching scripts instead of MPI applications.

- **Question:** The application runs but does not generate intermediate trace files (*.mpit)
  Answer 1: Check that environment variables are correctly passed to the application.
  Answer 2: If the code is Fortran, check that the number of underscores used to decorate routines in the instrumentation library matches the number of underscores added by the Fortran compiler you used to compile and link the application. You can use the `nm` and `grep` commands to check it.

- **Question:** The instrumentation begins for a single process instead for several processes?
  Answer 1: Check that you place the appropriate parameter to indicate the number of tasks (typically -np).
  Answer 2: Some MPI implementation require the application to receive special MPI parameters to run correctly. For example, MPICH based on CH-P4 device require the binary to receive som paramters. The following example is an sh-script that solves this issue:
  ```
  #!/bin/sh
  EXTRAE_CONFIG_FILE=extrae.xml ./mpi_program $@ real_params
  ```

- **Question:** The application blocks at the beginning?
  Answer : The application may be waiting for all tasks to startup but only some of them are running. Check for the previous question.

- **Question:** The resulting traces does not contain the routines that have been instrumented.
  Answer 1: Check that the routines have been actually executed.
  Answer 2: Some compilers do automatic inlining of functions at some optimization levels (e.g., Intel Compiler at -O2). When functions are inlined, they do not have entry and exit blocks and cannot be instrumented. Turn off inlining or decrease the optimization level.

- **Question:** Number of threads = 1?
  Answer : Some MPI launchers (*i.e.* mpirun, poe, mprun...) do not export all the environment variables to all tasks. Look at chapter 7 to workaround this and/or contact your support staff to know how to do it.

- **Question:** When running the instrumented application, the loader complains about:
  `undefined symbol: clock_gettime`
  Answer : The instrumentation package was configured using `--enable-posix-clock` and on many systems this implies the inclusion of additional libraries (namely, `-lrt`).

C.3 Performance monitoring counters FAQ

- **Question:** How do I know the available performance counters on the system?
  Answer 1: If using PAPI, check the `papi_avail` or `papi_native_avail` commands found in the PAPI installation directory.
  Answer 2: If using PMAPI (on AIX systems), check for the `pmlist` command. Specifically, check for the available groups running `pmlist -g -1`.

- **Question:** How do I know how many performance counters can I use?
  Answer: The Extrae package can gather up to eight (8) performance counters at the same time. This also depends on the underlying library used to gather them.

- **Question:** When using PAPI, I cannot read eight performance counters or the specified in `papi_avail` output.
  Answer 1: There are some performance counters (those listed in `papi_avail`) that are classified as derived. Such performance counters depend on more than one counter increasing the number of real performance counters used. Check for the derived column within the list to check whether a performance counter is derived or not.
  Answer 2: On some architectures, like the PowerPC, the performance counters are grouped in a such way that choosing a performance counter precludes others from being elected in the same set. A feasible work-around is to create as many sets in the XML file to gather all the required hardware counters and make sure that the sets change from time to time.

C.4 Merging traces FAQ

- **Question:** The `mpi2prv` command shows the following messages at the start-up:
  `PANIC! Trace file TRACE.0000011148000001000000.mpit is 16 bytes too big!`
  `PANIC! Trace file TRACE.0000011147000002000000.mpit is 32 bytes too big!`
  `PANIC! Trace file TRACE.0000011146000003000000.mpit is 16 bytes too big!`
  and it dies when parsing the intermediate files.
  Answer 1: The aforementioned messages are typically related with incomplete writes in disk. Check for enough disk space using the `quota` and `df` commands. **Answer 2:** If your system supports multiple ABIs (for example, linux x86-64 supports 32 and 64 bits ABIs), check that the ABI of the target application and the ABI of the merger match.

- **Question:** The resulting Paraver tracefile contains invalid references to the source code.
  Answer: This usually happens when the code has not been compiled and linked with the -g flag. Moreover, some high level optimizations (which includes inlining, interprocedural analysis, and so on) can lead to generate bad references.

- **Question:** The resulting trace contains information regarding the stack (like callers) but their value does not coincide with the source code.
  Answer: Check that the same binary is used to generate the trace and referenced with the the `-e` parameter when generating the Paraver tracefile.

Appendix D

Instrumented routines

D.1 Instrumented MPI routines

These are the instrumented MPI routines in the Extrae package:

- MPI_Init
- MPI_Init_thread[1]
- MPI_Finalize
- MPI_Bsend
- MPI_Ssend
- MPI_Rsend
- MPI_Send
- MPI_Bsend_init
- MPI_Ssend_init
- MPI_Rsend_init
- MPI_Send_init
- MPI_Ibsend
- MPI_Issend
- MPI_Irsend
- MPI_Isend
- MPI_Recv
- MPI_Irecv
- MPI_Recv_init

- MPI_Reduce
- MPI_Reduce_scatter
- MPI_Allreduce
- MPI_Barrier
- MPI_Cancel
- MPI_Test
- MPI_Wait
- MPI_Waitall
- MPI_Waitany
- MPI_Waitsome
- MPI_Bcast
- MPI_Alltoall
- MPI_Alltoallv
- MPI_Allgather
- MPI_Allgatherv
- MPI_Gather
- MPI_Gatherv
- MPI_Scatter
- MPI_Scatterv
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Comm_create
- MPI_Comm_dup
- MPI_Comm_split
- MPI_Cart_create
- MPI_Cart_sub
- MPI_Start
- MPI_Startall

- MPI_Request_free
- MPI_Scan
- MPI_Sendrecv
- MPI_Sendrecv_replace
- MPI_File_open[2]
- MPI_File_close[2]
- MPI_File_read[2]
- MPI_File_read_all[2]
- MPI_File_write[2]
- MPI_File_write_all[2]
- MPI_File_read_at[2]
- MPI_File_read_at_all[2]
- MPI_File_write_at[2]
- MPI_File_write_at_all[2]
- MPI_Get[3]
- MPI_Put[3]

D.2 Instrumented OpenMP runtimes

D.2.1 Intel compilers - icc, iCC, ifort

The instrumentation of the Intel OpenMP runtime for versions 8.1 to 10.1 is only available using the Extrae package based on DynInst library.

These are the instrument routines of the Intel OpenMP runtime functions using DynInst:

- __kmpc_fork_call
- __kmpc_barrier
- __kmpc_invoke_task_func
- __kmpc_set_lock[4]
- __kmpc_unset_lock[4]

[1]The MPI library must support this routine
[2]The MPI library must support MPI/IO routines
[3]The MPI library must support 1-sided (or RMA -remote memory address-) routines

The instrumentation of the Intel OpenMP runtime for version 11.0 to 12.0 is only available using the Extrae package based on the `LD_PRELOAD` mechanism. The instrumented routines include:

- __kmpc_fork_call
- __kmpc_barrier
- __kmpc_dispatch_next_4
- __kmpc_dispatch_next_8
- __kmpc_single
- __kmpc_end_single
- __kmpc_critical[4]
- __kmpc_end_critical[4]
- omp_set_lock[4]
- omp_unset_lock[4]

D.2.2 IBM compilers - xlc, xlC, xlf

Extrae supports IBM OpenMP runtime 1.6.

These are the instrumented routines of the IBM OpenMP runtime:

- _xlsmpParallelDoSetup_TPO
- _xlsmpParRegionSetup_TPO
- _xlsmpWSDoSetup_TPO
- _xlsmpBarrier_TPO
- _xlsmpSingleSetup_TPO
- _xlsmpWSSectSetup_TPO
- _xlsmpRelDefaultSLock[4]
- _xlsmpGetDefaultSLock[4]

[4]The instrumentation of OpenMP locks can be enabled/disabled

D.2.3 GNU compilers - gcc, g++, gfortran

Extrae supports GNU OpenMP runtime 4.2.

These are the instrumented routines of the IBM OpenMP runtime:

- GOMP_parallel_start
- GOMP_parallel_sections_start
- GOMP_parallel_end
- GOMP_sections_start
- GOMP_sections_next
- GOMP_sections_end
- GOMP_sections_end_nowait
- GOMP_loop_end
- GOMP_loop_end_nowait
- GOMP_loop_static_start
- GOMP_loop_dynamic_start
- GOMP_loop_guided_start
- GOMP_loop_runtime_start
- GOMP_parallel_loop_static_start
- GOMP_parallel_loop_dynamic_start
- GOMP_parallel_loop_guided_start
- GOMP_parallel_loop_runtime_start
- GOMP_loop_static_next
- GOMP_loop_dynamic_next
- GOMP_loop_guided_next
- GOMP_loop_runtime_next
- GOMP_barrier
- GOMP_critical_name_enter[4]
- GOMP_critical_name_exit[4]
- GOMP_critical_enter[4]
- GOMP_critical_exit[4]
- GOMP_atomic_enter[4]
- GOMP_atomic_exit[4]

D.3 Instrumented pthread runtimes

These are the instrumented routines of the pthread runtime:

- pthread_create
- pthread_detach
- pthread_join
- pthread_mutex_lock
- pthread_mutex_trylock
- pthread_mutex_timedlock
- pthread_mutex_unlock
- pthread_rwlock_rdlock
- pthread_rwlock_tryrdlock
- pthread_rwlock_timedrdlock
- pthread_rwlock_wrlock
- pthread_rwlock_trywrlock
- pthread_rwlock_timedwrlock
- pthread_rwlock_unlock