SISPROFILING Library: User's Guide
Version 1.0
Incomplete documentation[1]

Hong-Linh Truong
Institute for Software Science
University of Vienna
Liechtensteinstr. 22, 1109 Vienna
AUSTRIA
truong@par.univie.ac.at

October 6, 2003

[1]The work described in this paper was partially supported by the Special Research Program SFB F1104 "AURORA" of the Austrian Science Fund.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SISPROFILING is a library that supports for measurement performance of distributed and parallel programs. The library can be used to capture execution behavior of C/Fortran OpenMP/MPI, HPF+ and hybrid programs that are running on various platforms such as Linux, Sun Solaris, IBM AIX, etc. SISPROFILING collects performance information of programs and stores them into files for post-mortem analysis or sends the data to online performance analysis tools. The current libraries allows to measure various performance information of code regions including arbitrary code regions:

- number of invocations of a code region
- timing metrics (such as wall clock time, user time, system time), and
- hardware metrics (the number of hardware metrics can be measured dependent on the individual platform)

SISPROFILING allows to measure multiple timing or/and hardware parameters at the same time. Hardware parameters based on PAPI[2][1]. Measurement process can be enabled or disabled by using APIs inserted into codes. Performance data is maintained in thread-based.

Chapter 2

Installation SISPROFILING

2.1 Supported Platforms

The current version can be used to measure C/Fortran OpenMP, MPI, OpenMPI as well as HPF+ (with VFC compiler). Supported platforms are:

1. Linux (2.2, 2.4)
2. Sun Solaris (2.6,2.7,2.8)
3. IBM AIX 5.0, 5.1

SISPROFILING is a portable library so that it can easily be compiled on other platforms. Due to the lack of available platforms this work has not been tested.

2.2 Configuration Options

After uncompressing and untarring SISPROFILING, the user needs to configure, compile and install the package. This is done by invoking:

```
$./configure [option]
```

SISPROFILING is configured by running the configure script with appropriate options that select the profiling and tracing components that are used to build the SISPROFILING library. The following command-line options are available to configure:

–prefix=<directory>

Description: Specifies the destination directory where the header, library and binary files are copied. By default, these are copied to subdirectories `bin`, `lib`, `include` in the SISPROFILING root directory.

–with-sistime

Description: This flag is used to configure timing metrics that will be measured by SISPROFILING. With this option, SISPROFILING will also measure *user time*, *system time*. Time will be measured in microsecond.

–with-sisrusage

Description: This option configures SISPROFILING to use *getrusage* function to measure *page faults*, *swaps* of programs.

–with-sisdebug

Description: This option is used to configure a debugged SISPROFILING library.

–with-c++=<C++ compiler>

Description: Specifies the name of the C++ compiler. Supported C++ compilers includes CC, g++ (from GNU), and pgCC (from PGI).

–with-cc=<C Compiler>

Description: Specifies the name of the C compiler. Supported C compilers include cc, gcc (from GNU), pgcc (from PGI).

–with-f77=<Fortran 77 compiler>

Description: Specifies the name of the Fortran 77 compiler. Supported F77 compilers include f77(GNU),pg77(PGI).

–with-f90=<Fortran f90 compiler>

Description:

–with-jdk=<jdk root dir>

Description: Gives the path of JDK distribution.

–with-mpi=<MPI top directory>

Description: Specifies the path of MPI library (e.g. MPICH, LAM).

–with-papi =<PAPI top directory>

Description: Specifies the directory where PAPI resides.

–with-papimultiplex

Description: Enables multiplex mode for PAPI so that multiple hardware counters can be measured at the same time.

–with-globus=<GLOBUS top directory>

Description: Specifies the path of GLOBUS toolkit. For more information of GLOBUS see http://www.globus.org/. This option is for features currently being implemented.

–with-flavor=<globus_flavor>

Description: Specifies flavor for Globus. This option is for features currently being implemented.

–with-dyninst=<top_dyninst_dir>

Description: Specifies dyninst library (www.dyninst.org).This option is for features currently being implemented.

2.3 Installation

After configuration the user can compile and install SISPROFILING library with the following command:

```
$make
$make install
```

SISPROFILING will be installed into the directory pointed by `-prefix`. If the user doesn't set this directory, SISPROFILING will be installed in the current directory. The following example shows how to install SISPROFILING on a Linux machine.

```
$./configure --prefix=/home/linh/profiletools/sisprofiling --with-cc=gcc
--with-f77=pgf77 --with-f90=pgf90 --with-sistime --with-sisrusage
--with-mpich=/opt/local/mpich --with-globus=/opt/local/globus-1.1.4
$make
$make install
```

Chapter 3

Instrumentation with SISPROFILING

3.1 Introduction

SISPROFILING is used by SCALEA framework [3] as its instrumentation library.By using SCALEA the user doesn't need to know how to instrument his program with SISPROFILING library. SCALEA Instrumentation System gets controls from the user and automatically inserts SISPROFILING APIs to source code. In this chapter we present SISPROFILING APIs for the user who wants to manually instrument his programs. The current SISPROFILING version provides APIs for C/Fortran OpenMP, MPI, OpenMPI and HPF+ (with VFC compiler).

There are three key concepts in order to work with SISPROFILING library:

- *Code regions*: every program consists of a set of code regions which can range from a single statement to the entire program unit. The user selects interesting code regions and instruments these code regions by inserting probes enclosing the code regions.
- *Code region types*: a selected code region can be a loop, subroutine call, arbitrary code region (consists a set of sequential statements),etc. SIS supports the programmer control instrumentation by defining predefined code regions. The predefined code regions are classified into generic and specific code regions. Generic code regions are loops, subroutine calls, etc. Specific code regions are HPF, OpenMP, MPI code regions.
- *Performance metrics*: what the user wants to measure are performance metrics of code regions. Performance metrics are classified into timing, hardware parameter and overhead metrics. While timing and hardware metrics can be provided by SISPROFILING directly, it is required further analysis in order to obtain overhead metrics (e.g. by SCALEA).

3.2 Predefined Code regions

Currently SIS supports more than 40 predefined code regions classified into generic code regions and specific code regions as shown in Fig. 3.1.

Table 3.1 shows types of predefined code regions currently supported by SIS.

Mnemonics	Descriptions
CR_P	Main program
CR_A	Arbitrary code region
CR_L	All loops (general)
CR_U	Outermost loop
CR_B	Branch code region
CR_W	I/O Write operation
CR_R	I/O read operation
CR_O	I/O open operation
CR_C	I/O close operation

Mnemonics	Descriptions
CR_Y	procedure's internal (function or subroutine)
CR_S	Subroutine call
CR_F	Function calls
CR_COMALL	All common code regions
CR_I	INDEPENDENT loop (HPF)
CR_D	Work distribution (HPF)
CR_N	Inspector (HPF)
CR_X	Executor (HPF)
CR_G	Gather (HPF)
CR_T	Scatter (HPF)
CR_HPFALL	all HPF code regions
CR_OMPPA	OMP PARALLEL
CR_OMPPD	OMP PARALLEL DO
CR_OMPPS	OMP PARALLEL SECTIONS
CR_OMPPW	OMP PARALLEL WORKSHARE
CR_OMPDO	OMP DO
CR_OMPSE	OMP SECTIONS
CR_OMPWO	OMP WORKSHARE
CR_OMPSI	OMP SINGLE
CR_OMPMA	OMP MASTER
CR_OMPBA	OMP BARRIER
CR_OMPCR	OMP CRITICAL
CR_OMPAT	OMP ATOMIC
CR_OMPOR	OMP ORDERED
CR_OMPFL	OMP FLUSH
CR_OMPSE	OMP SECTION
CR_OMPICR	Codes inside OMP CRITICAL
CR_OMPIOR	Codes inside OMP ORDERED
CR_OMPISE	Codes inside OMP SINGLE
CR_OMPBPA	OMP PARALLEL directive
CR_OMPEPA	OMP END PARALLEL directive
CR_OMPIDO	the body of do loop of OMP DO
CR_OMPLO	openmp locks
CR_OMPALL	All OpenMP code regions
CR_MPISTARTUP	MPI Initialization and MPI finalization
CR_MPIP2P	Point-to-point communication
CR_MPISEND	Send
CR_MPIRECV	Receive
CR_MPICOL	Collective communication
CR_MPITP	MPI data type conversions
CR_MPIBA	MPI barrier
CR_MPIALL	All MPI code regions
CR_OTHERREP	Replicated code region
CR_OTHERSEQ	Sequential code region

Table 3.1: Code region Mnemonics

3.3 Performance metrics

When instrumenting a program, the user wants to collect some performance metrics that are relevant to the program. Such performance metrics are dependent on the user's purpose. In some cases, the user wants to select many performance metrics and tools may not fulfill the requirements of the user.

An instrumentation library may not provide some performance metrics directly from measurement process.

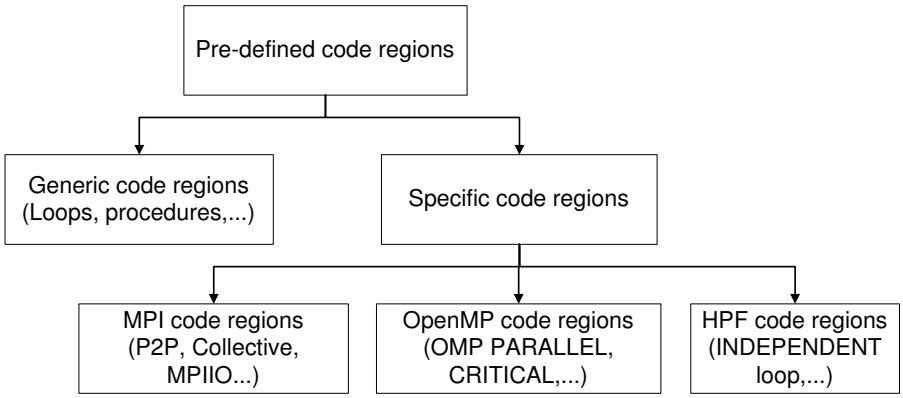

Figure 3.1: Classification of Code Regions in SCALEA

Mnemonics	**Descriptions**
WTIME	Wall clock time
UTIME	User time
STIME	System time
CTIME	CPU Time, include user time and system time
ATIME	All above timing metrics

Table 3.2: Mnemonics for timing metrics

Instead, it is required further analysis in order to obtain such performance metrics. SISPROFILING provides more than one hundred performance metrics which are categorized into three classes: timing, hardware parameter metrics. Timing metrics are timing quantities of code regions such as wall clock time, user time, etc. Hardware metrics are hardware counter quantities of code regions such as floating point operation, cache misses, etc. The other performance metrics (e.g overhead metrics) can be obtained by further analysis. Table 3.2 shows timing metrics whereas hardware parameter metrics are described in table 3.3. In the current implementation of SISPROFILING, the user can select hardware metrics that he wants to measure by using APIs or environment variables. Timing metrics are fixed by SISPROFILING.

Mnemonics	**Descriptions**
MAJFLT	page faults requiring physical I/O
MINFLT	page faults not requiring physical I/O
NSWAP	Number of swap
L1_DCM	Level 1 data cache misses
L1_ICM	Level 1 instruction cache misses
L2_DCM	Level 2 data cache misses
L2_ICM	Level 2 instruction cache misses
L3_DCM	Level 3 data cache misses
L3_ICM	Level 3 instruction cache misses
L1_TCM	Level 1 cache misses
L2_TCM	Level 2 cache misses
L3_TCM	Level 3 cache misses
CA_SNP	Requests for a snoop
CA_SHR	Requests for exclusive access to shared cache line
CA_CLN	Requests for exclusive access to clean cache line
CA_INV	Requests for cache line invalidation
CA_ITV	Requests for cache line intervention
L3_LDM	Level 3 load misses

Mnemonics	Descriptions
L3_STM	Level 3 store misses
BRU_IDL	Cycles branch units are idle
FXU_IDL	Cycles integer units are idle
FPU_IDL	Cycles floating point units are idle
LSU_IDL	Cycles load/store units are idle
TLB_DM	Data translation lookaside buffer misses
TLB_IM	Instruction translation lookaside buffer misses
TLB_TL	Total translation lookaside buffer misses
L1_LDM	Level 1 load misses
L1_STM	Level 1 store misses
L2_LDM	Level 2 load misses
L2_STM	Level 2 store misses
BTAC_M	Branch target address cache misses
PRF_DM	Data prefetch cache misses
L3_DCH	Level 3 data cache hits
TLB_SD	Translation lookaside buffer shootdowns
CSR_FAL	Failed store conditional instructions
CSR_SUC	Successful store conditional instructions
CSR_TOT	Total store conditional instructions
MEM_SCY	Cycles Stalled Waiting for memory accesses
MEM_RCY	Cycles Stalled Waiting for memory Reads
MEM_WCY	Cycles Stalled Waiting for memory writes
STL_ICY	Cycles with no instruction issue
FUL_ICY	26 Cycles with maximum instruction issue
STL_CCY	Cycles with no instructions completed
FUL_CCY	Cycles with maximum instructions completed
HW_INT	Hardware interrupts
BR_UCN	Unconditional branch instructions
BR_CN	Conditional branch instructions
BR_TKN	Conditional branch instructions taken
BR_NTK	Conditional branch instructions not taken
BR_MSP	Conditional branch instructions mispredicted
BR_PRC	Conditional branch instructions correctly predicted
FMA_INS	FMA instructions completed
TOT_IIS	Instructions issued
TOT_INS	Instructions completed
INT_INS	Integer instructions
FP_INS	Floating point instructions
LD_INS	Load instructions
SR_INS	Store instructions
BR_INS	Branch instructions
VEC_INS	Vector/SIMD instructions
FLOPS	Floating point instructions per second
RES_STL	Cycles stalled on any resource
FP_STAL	Cycles the FP unit(s) are stalled
TOT_CYC	Total cycles
IPS	Instructions per second
LST_INS	Load/store instructions completed
SYC_INS	Synchronization instructions completed
L1_DCH	Level 1 data cache hits
L2_DCH	Level 2 data cache hits
L1_DCA	Level 1 data cache accesses
L2_DCA	Level 2 data cache accesses
L3_DCA	Level 3 data cache accesses

Mnemonics	Descriptions
L1_DCR	Level 1 data cache reads
L2_DCR	Level 2 data cache reads
L3_DCR	Level 3 data cache reads
L1_DCW	Level 1 data cache writes
L2_DCW	Level 2 data cache writes
L3_DCW	Level 3 data cache writes
L1_ICH	Level 1 instruction cache hits
L2_ICH	Level 2 instruction cache hits
L3_ICH	Level 3 instruction cache hits
L1_ICA	Level 1 instruction cache accesses
L2_ICA	Level 2 instruction cache accesses
L3_ICA	Level 3 instruction cache accesses
L1_ICR	Level 1 instruction cache reads
L2_ICR	Level 2 instruction cache reads
L3_ICR	Level 3 instruction cache reads
L1_ICW	Level 1 instruction cache writes
L2_ICW	Level 2 instruction cache writes
L3_ICW	Level 3 instruction cache writes
L1_TCH	Level 1 total cache hits
L2_TCH	Level 2 total cache hits
L3_TCH	Level 3 total cache hits
L1_TCA	Level 1 total cache accesses
L2_TCA	Level 2 total cache accesses
L3_TCA	Level 3 total cache accesses
L1_TCR	Level 1 total cache reads
L2_TCR	Level 2 total cache reads
L3_TCR	Level 3 total cache reads
L1_TCW	Level 1 total cache writes
L2_TCW	Level 2 total cache writes
L3_TCW	Level 3 total cache writes
FML_INS	Floating point multiply instructions
FAD_INS	Floating point add instructions
FDV_INS	Floating point divide instructions
FSQ_INS	Floating point square root instructions
FNV_INS	Floating point inverse instructions

Table 3.3: Mnemonics for hardware metrics

3.4 Programming APIs

3.4.1 C/C++ APIs

SISPROFILING provides include files that declare SISPROFILING APIs and constant variables.

- **sisinst.h** is include file declaring C/C++ APIs of SISPROFILING library (*sisprofile.h* in previous version).
- **siscr.h**: the include file declares types of predefined code regions.

In the following, we describe main APIs of SISPROFILING.

void sis_init()

Descriptions: This function is used to initialize the SISPROFILING library. It must be called in the master thread before any other SISPROFILING APIs.

Errors: If this function generates an error, the SISPROFILING library could not initialize and then the measurement will not work.

Returns: None

void sis_set_node(int processid)

Descriptions: This function is used to set process identifier for an executing process. Each process of application has a different identifier. Notice that the process's identifier isn't the same as the process id provided by the operating system. The *processid* is used in SISPROFILING library to maintain performance data for each process separately. Therefore its value ranges normally from 0 to max-1 where is max is number of executing processes of the application. If this function isn't called, the process's identifier will be 0.

Errors: None.

Returns: None.

void sis_par_set_node()

Descriptions: This function is used to set the *process identifier* for a MPI/OpenMPI program. It will automatically assign an identifier for a process. These identifiers distinguish between processes of the application. This function is only applied for MPI/OpenMPI programs and it must be called after MPI_Init(). The process identifier is used in SISPROFILING library to maintain performance data for each process separately. If this function is not called, the process identifier will be 0. Notice that if you use MPI wrapper library, you don't need to call this function.

Errors: None.

Returns: None.

void sis_start(int mpt_id, int mpt_group)

Descriptions: This function is used to start a probe measuring a code region. Running threads that encounter this function will execute it. The *mpt_id* is the identifier of code region which is unique for each code region. The *mpt_group* is type of predefined code regions which is defined in the include file.

Errors: If this function generates an error, this code region will not be measured.

Returns: None.

void sis_stop(int mpt_id)

Descriptions: This function is used to stop the probe with the id *mpt_id*. Threads that encounter this function will execute the function. The function is used in conjunction with the *sis_start()* function. Notice that pairs of probes can be nested but not overlapping.

Errors: If this function generates an error, the measurement will not be stopped.

Returns: None.

Examples:

```
sis_start(1,CR_L);
for (i=0; i <100 ;i++){
 ...
}
sis_stop(1);
```

void sis_master_start(int mpt_id, int mpt_group)

Descriptions: The same as the *sis_start()* function but it only starts the probes in the master thread.

Usages:

Errors: If this function generates an error, this code region will not be measured.

Return values: None.

void sis_master_stop(int mpt_id)

Descriptions: The same as the *sis_stop()* function but it only stops the probes in the master thread. The function is used in conjunction with the *sis_master_start()* function. Notice that pairs of probes can be nested but not overlapping.

Errors: If this function generates an error, the measurement will not be stopped.

Return values: None.

Examples:

```
sis_start(1,SIS_L);
for (i=0; i <100 ;i++) {
 ...
}
sis_stop(1);
```

void sis_exit(void);

Description: this function is used to release SISPROFILING library. Profiling data will be saved into files.

Errors:

Returns: None.

void sis_enable_instrument(void)

Descriptions: This function is used to enable the instrumentation process that has been disabled by *sis_disable_instrumen*
Every threads will enable their instrumentation process. See *sis_disable_instrument()* for detail information.

Errors:

Returns: None.

void sis_disable_instrument(void)

Descriptions: This function is used to disable the instrumentation process. In this mode, a probe will not be started to measure and all probes that are currently executing will be suspended. This function is applied for thread based.

Errors:

Returns: None.

void sis_hardwaremetricsset(char *hardwarename)

void sis_timingmetricsset(char *timingname)

void sis_metrics_unset(char *metricname)

3.4.2 Fortran APIs

SISPROFILING provides include files that declare constant variables for Fortran.

- **fsiscr.h** declares types of predefined code regions for Fotran 77.
- **siscr.f90** declares types of predefined code regions for Fotran 90 programs.
- **sisinst.f90** declares functions used to instrument Fortran 90 programs.

```
include ‘‘fsiscr.h’’
or
use siscr
use sisinst
```

The SISPROFILING APIs for Fortran are similar to the ones in C/C++. Here is a list of Fortran APIs:

sis_init()

sis_exit()

sis_set_node(integer nodeid);

sis_par_set_node();

sis_start(integer mpt_id, integer mpt_group) ;

sis_stop(integer mpt_id) ;

sis_master_start(integer mpt_id, integer mpt_group) ;

sis_master_stop(integer mpt_id) ;

sis_enable_instrument()

sis_disable_instrument()

sis_hardwaremetrics_set(character name)

sis_timingmetrics_set(character name)

sis_metrics_unset(character name)

Chapter 4

Compiling, Linking and Executing Instrumented Program

4.1 Introduction

After instrumenting a program automatically or manually, instrumented source files are compiled and linked. When building the application successfully, the application can be run on the chosen platform. The following steps are proposed for building applications:

1. Copy/create the suitable SIS Makefile
2. Compile and link the instrumented programs.
3. Run the application.

whereas the the first step needs to be done only once. If the program is changed, only the last two steps are to be repeated.

4.2 Compiling and Linking

This section describes how to build an instrumented programs. SISPROFILING instrumentation library contains following libraries:

- Profile libraries
  - libsisseq.a : library for sequential programs.
  - libsismpi.a: library for MPI programs
  - libsisomp.a: library for OpenMP program
  - libsisopenmpi.a library for OpenMP/MPI programs
  - libsismpiwrapper.a: MPI wrapper library
- Trace libraries
  will be added soon
- Online measurement libraries
  will be added soon

To compile and link instrumented code with SISPROFILING instrumentation library, firstly the user chooses the libraries suitable for his/her application and builds the application. Note that SISPROFILING libraries use PAPI library for measuring hardware parameter measurement so that SISPROFILING installed with hardware parameter measurement mode needs to be linked with PAPI library.

4.2.1 Choose SISPROFILING libraries for your applications

4.2.1.1 Sequential programs

The SISPROFILING sequential library is *libsisseq.a*. If SISPROFILING is installed to work with PAPI, PAPI library must be linked after SISPROFILING library. Here is a sample of Makefile.

```
SISPROFILINGDIR = /home/linh/profiletools/scalea/lib
SISINCDIR       = /home/linh/profiletools/scalea/include
PAPILIB = /opt/local/PAPI/lib/libpapi.a
LIBS          =  -L$(SISPROFILINGDIR) -lsisseq /opt/local/PAPI/lib/libpapi.a
FFLAGS = -I$(SISINCDIR)
LDFLAGS         =
RM              = /bin/rm -f
F90             = pgf90
TARGET          = md_sis
all:            $(TARGET)
install:        $(TARGET)
$(TARGET):      $(TARGET).o
        $(F90) $(LDFLAGS) $(TARGET).o  -o $@ $(LIBS)
$(TARGET).o : $(TARGET).f90
$(F90) $(FFLAGS) -c $(TARGET).f90
clean:
$(RM) $(TARGET).o $(TARGET)
```

4.2.1.2 OpenMP programs

The SISPROFILING OpenMP library is *libsisomp.a*. If SISPROFILING is installed to work with PAPI, PAPI library must be linked after SISPROFILING library. Here is a sample of Makefile.

```
SISPROFILINGDIR = /home/linh/profiletools/scalea/lib
SISINCDIR       = /home/linh/profiletools/scalea/include
PAPILIB = /opt/local/PAPI/lib/libpapi.a
LIBS          =  -L$(SISPROFILINGDIR) -lsisomp $(PAPILIB)
FFLAGS = -I$(SISINCDIR)
LDFLAGS         =
RM              = /bin/rm -f
F90             = pgf90 -mp
TARGET          = md_sis
all:            $(TARGET)
install:        $(TARGET)
$(TARGET):      $(TARGET).o
        $(F90) $(LDFLAGS) $(TARGET).o  -o $@ $(LIBS)
$(TARGET).o : $(TARGET).f90
$(F90) $(FFLAGS) -c $(TARGET).f90
clean:
$(RM) $(TARGET).o $(TARGET)
```

4.2.1.3 OpenMPI programs

The SISPROFILING OpenMPI library is *libsisopenmpi.a*. If SISPROFILING is installed to work with PAPI, PAPI library must be linked after SISPROFILING library. Optionally, libmpiwrapper.a for measure MPI routines via wrapper mechanism could be used. Here is a sample of Makefile.

```
SISPROFILINGDIR = /home/linh/profiletools/scalea/lib
SISINCDIR       = /home/linh/profiletools/scalea/include
MPILIBDIR= -L/usr/local/pgi/linux86/lib
PAPILIB = /opt/local/PAPI/lib/libpapi.a
LIBS = -L$(MPILIBDIR) -lfmpich -L$(SISPROFILINGDIR) -lsismpiwrapper -lsisopenmpi $(PAPILIB)
-L$(MPILIBDIR) -lpmpich -lmpich
FFLAGS = -I$(SISINCDIR)
```

```
LDFLAGS        =
RM             = /bin/rm -f
F90            = pgf90
TARGET         = md_sis
all:           $(TARGET)
install:       $(TARGET)
$(TARGET):     $(TARGET).o
	$(F90) $(LDFLAGS) $(TARGET).o  -o $@ $(LIBS)
$(TARGET).o : $(TARGET).f90
$(F90) $(FFLAGS) -c $(TARGET).f90
clean:
$(RM) $(TARGET).o $(TARGET)
```

4.2.1.4 MPI programs

The SISPROFILING MPI library is *libsismpi.a*. If SISPROFILING is installed to work with PAPI, PAPI library must be linked after SISPROFILING library. Optionally, libmpiwrapper.a for measure MPI routines via wrapper mechanism can be used. Here is a sample of Makefile.

```
SISPROFILINGDIR = /home/linh/profiletools/scalea/lib
SISINCDIR      = /home/linh/profiletools/scalea/include
MPILIBDIR= -L/usr/local/pgi/linux86/lib
PAPILIB = /opt/local/PAPI/lib/libpapi.a
LIBS = -L$(MPILIBDIR) -lfmpich -L$(SISPROFILINGDIR) -lsismpiwrapper -lsismpi $(PAPILIB) -
L$(MPILIBDIR) -lpmpich -lmpich
FFLAGS = -I$(SISINCDIR)
LDFLAGS        =
RM             = /bin/rm -f
F90            = pgf90
TARGET         = md_sis
all:           $(TARGET)
install:       $(TARGET)
$(TARGET):     $(TARGET).o
	$(F90) $(LDFLAGS) $(TARGET).o  -o $@ $(LIBS)
$(TARGET).o : $(TARGET).f90
$(F90) $(FFLAGS) -c $(TARGET).f90
clean:
$(RM) $(TARGET).o $(TARGET)
```

4.2.1.5 HPF and HPF/OpenMP program compiled by VFC compiler

VFC compiles HPF, HPF/OpenMP programs to MPI and OpenMPI programs respectively. After that the user has to link the output programs with VFC RTS (VFC runtime system). Therefore SISPROFILING libraries for HPF and HPF/OpenMP program are the same as the ones for MPI and OpenMPI program. If SISPROFILING is installed to work with PAPI, PAPI library must be linked after SISPROFILING library. Here is a sample of Makefile for a HPF/OpenMP program.

```
#### Makefile for compiling and linking vfc-files on gescher  ####

T       = bwomp
#
CXX      = pgCC
CC       = pgcc
CLINKER  = pgcc
F90      = pgf90
FLINKER  = pgf90
AR       = ar
#
RTS      = /home/sipka/vfc/rts
RTS_LIB  = $(RTS)/lib
```

```
RTS_INC   = $(RTS)/include
#
RD_LIB    = $(RTS)/rd/lib
PARS_LIB  = $(RTS)/pars/lib
PARIO_LIB = $(RTS)/pario/lib
AD_LIB    = $(RTS)/adlib/adlib_1.0/lib
#
PG_DIR   = /opt/local/pgi/linux86
PG_LIB   = $(PG_DIR)/lib
PG_INC   = $(PG_DIR)/include
#
STD_DIR   = /usr
STD_LIB   = $(STD_DIR)/lib
#
ARCH      = LINUX
COMM      = ch_p4
#
MPI_DIR   = /opt/local/mpich
MPI_INC   = $(MPI_DIR)/include
MPI_LIB   = $(MPI_DIR)/lib
#SIS
SISPROFILINGDIR = /home/linh/profiletools/scalea/lib
SISINCDIR       = /home/linh/profiletools/scalea/include
PAPILIB = /opt/local/PAPI/lib/libpapi.a
SISLIB= -L$(MPI_LIB) -fmpich -L$(SISPROFILINGDIR) -lsismpiwrapper -lsismpi $(PAPILIB)
#
OPTFLAGS  = -v -mp
FFLAGS    = $(SISINCDIR) $(OPTFLAGS) -Minfo -I$(RTS_INC) -I$(PG_INC)
MPI_FLAGS = -lpmpich -lmpich
PG_FLAGS  = -lpgf90 -lpgf90rtl -lpgf90_rpm1 -lpgf902 -lstd -lC -lpgmp \
          -lpgthread -lpgc
FLAGS     = -mp -lc
LIBS      = $(SISLIB) -L$(MPI_LIB) $(MPI_FLAGS) -L$(PG_LIB) $(PG_FLAGS) \
          -L$(STD_LIB) $(FLAGS)

### End User configurable options ###
EXECS = $(T)

default: $(EXECS)

$(T): $(T).o
       $(FLINKER) $(OPTFLAGS) -o $(T) $(T).o    \
         -L$(RTS_LIB) -lpario -lrd -lvim -lpars -lrd -lpars \
         -L$(AD_LIB) -lvfcad -ladlib  \
         $(LIBS)

$(T).o: $(T)_vfc.f90
       $(F90) $(FFLAGS) -c -o $(T).o $(T)_vfc.f90

clean:
       /bin/rm -f *.o *~ $(EXECS)
```

4.3 Executing instrumented programs

Before executing a program, some environment variables have to be set up. After that, the program is executed as the normal way.

4.3.1 Setting up environment variables for SISPROFILING

Setting up environment variables when using SISPROFILING library:

- **SISPROFILEDIR**: this environment variable indicates the directory where SISPROFILING files will be saved. For examples:

```
$setenv SISPROFILEDIR /home/linh/data
```

- **SISHWMETRIC[*num*]**: to set the hardware counter you want to measure. The value of *num* is from 1 to 16. For examples:

```
$setenv SISHWMETRIC1 FP_INS
$setenv SISHWMETRIC2 TOT_CYC
```

  SISPROFILING will measure two hardware parameters: total floating point instructions and total cycles.

- **SISHWENABLE [ON]**: this variable is used to enable or disable the hardware parameter measurement mode. If this variable is set ON, SISPROFILING will turn on hardware parameter measurement mode. And then the hardware metrics are set by SISHWMETRIC variable. If this variable isn't set ON, SISPROFILING will measure hardware parameters that are set by SISPROFILING APIs(see SISPROFILING APIs) and ignore SISHWMETRIC variables.

```
$setenv SISHWENABLE ON
```

- **SISXML**: to select wherether profile files will be outputed in XML format. If this variable is set, the profile files will be saved in XML format.

```
$setenv SISXML
```

4.3.2 SISPROFILING profile files

SISPROFILING profile files have the format of name as follows

- *sisprofile.[computationalnode].[processid].[threadid].log* where *computationalnode* is the name of the manchine where the process is executed, *processid* is the id of the process and *threadid* is the id of thread. These profile files are saved in the directory indicated by the environment variable **SISPROFILEDIR** or current working directory. Notice that the value of *processid* is not the same as the process id assigned by the Operating System.

SISPROFILING profile files are text files. Profiling and counting information of each thread are maintained separately and saved into a file. The internal format contains fields shown in Table 4.1.

The records of a SISPROFILING file are separated by spaces. The first row of a SISPROFILING file is a comment line which shows the meaning of columns in this file. Here is sample of SISPROFILING file:

```
#id parent group calls subs wtime utime stime majflt minflt nswap
1 0 1 1 3650 52134094 4714000 3450000 26643 210 0
31 1 21 3650 10950 48079874 45260000 2830000 2 0 0
33 31 25 3650 1230050 44496438 41670000 2810000 0 0 0
```

4.3.2.1 SISPROFILING xml profile file format

To be added

Field number	Description
1	probe id
2	parent probe id
3	probe group
4	number of sub probe calls
5	wall clock time(usec)
6	user time (usec)
7	system time(usec)
8,9	fage faults
10	number of process swaps
11,12,...	hardware counter

Table 4.1: Internal structure of SISPROFILING profile file

Chapter 5

Utilities working with SISPROFILING profile files

5.1 Introduction

5.2 ScaleaProfile

ScaleaProfile is a Java utility running in text mode that analyzes profile/trace files.

```
Usage:
        java scalea.tool.ScaleaProfile [-help] [-df description file]
                [-dprof files directory] [-toxml [xmlfilename]]
```

-help This option is used to show the help usage.

-df This option is used to set the *description file* of the current version. The *description file* is generated by SCALEA Instrumentation System when instrumenting source files. If the source files are instrumented manually, the *description file* is omitted.

-dprof . This option is used to set the directory in which profile/trace files are saved in post-mortem analysis. This utility will read profile/trace files from the directory set by the option. Notice that only files have names in format *sisprofile.node.processid.threadid.log* to be considered for analysis.

-toxml . This option is used to set the output mode. By default, the result is outputed to standard output in simple text rows. With this option, the output will be stored the file pointed by *xmlfilename* in XML format.

The following example presents the usage of *ScaleaProfile* to analysis performance of the experiment whose data are stored in /home/linh/projects/stommel/runp2xt4 :

```
java scalea.tool.ScaleaProfile -dprof /home/linh/projects/stommel/runp2xt4
```

5.3 Raw data to database

The *raw data to data repository* function is used to extract, filter relevant data from profile/trace files and build the dynamic code region tree call graph(DRG). Both filtered data and the DRG are then stored in the data repository. This function can be conducted through the SCALEA visualization. Moreover, a stand-alone program implemented this function is provided.

Figure 5.1 presents the *raw data to data repository* function that can be invoked through the **Preprocessing->Rawdata To Db** menu.

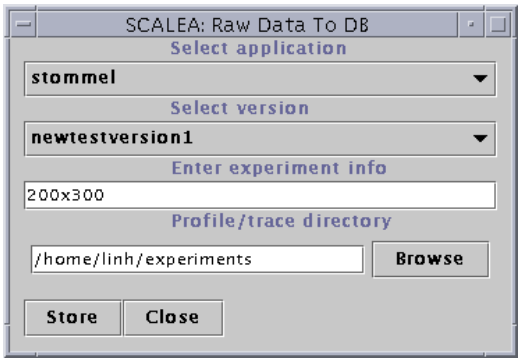

Figure 5.1: Raw data to Data Repository Window

The program *RawDataToDb* is a stand-alone program that implemented the *raw data to data repository* function. The *RawDataToDb* is written in Java and can be accessed via SCALEA Analysis and Visualization System package under directory *scalea.tool*. The usage of *RawDataToDb* program is shown as follows:

```
Usage:

java scalea.tool.RawDataToDb [-help] [-host dbserver] [-port port]
        [-user username] [-pass password] [-db databasename]
        [-dprof files directory] [-app "application info"]
        [-ver "version info"] [-exp "experiment info"]
```

-help This option is used to show the help usage.

-host Specifies the host name of the machine on which the database server is running.

-port Specifies the TCP/IP port on which the databaser server is listening for connections.

-user Connects to the database as the user *username*.

-pass Specifies the password of the corresponding user.

-db Specifies the name of the database to connect to.

-dprof Specifies the directory in which profile/trace files are saved.

-app Specifies the name of the application.

-ver Specifies the info of the version version. The application name and version information forms a unique tuple for each version. This information is used to determine a version that the source codes belong to. If the version is not available in the database, a new version will be created.

-exp Specifies the info of the current experiment.

The following example presents how to store raw data of an openmpi application into the database:

```
java scalea.tool.RawDataToDb -host gescher -user linh -db test -app "openmpi testing"
-ver "test1" -exp "2processes and 4 threads" -dprof /home/linh/projects/stommel/runp2xt4
```

Bibliography

[1] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceeding SC'2000*, November 2000.

[2] PAPI Project. http://icl.cs.utk.edu/projects/papi.

[3] Hong-Linh Truong and Thomas Fahringer. SCALEA Version 1.0: User's guide. Technical report, Institute for Software Science, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria, April 2001.