Bluebee toolchain pragma and platform description

BlueBee Multicore Technologies

CHAPTER 1

Pragma support

The pragmas supported relate to two aspects:

- Mapping
- Parallelism

We will describe each of them in detail in the next sections..

1.1 Controlling Mapping

Users can control mapping by annotating the code through a mapping #pragma on top of a function declaration (which affects all calls to that function), or on top a particular function call. A list of supported mapping annotations is presented in Table 1.1.

Note that the ”arg” clause can be combined with the ”call hw” pragma. Such an example is presented in Figure 1.1.

This means that with a simple mapping pragma, a function can be offloaded remotely from the GPP(general purpose processor), for example ARM

Table 1.1: List of pragmas

#pragma	Description
map call hw [FPGA|DSP] <instance id>	maps the function call to the [FPGA|DSP] <instance id>
map arg(argn,size) [arg(argm,sizem)...]	the pointer argn has size size

Figure 1.1: Mapping example

```
float *a, *b;

#pragma map call_hw PE 0
foo();
/* force foo to be mapped to PE instance 0 */
#pragma map call_hw GPP 0
bar();
/* force bar to be mapped to GPP instance 0 */
#pragma map call_hw PE 0 arg(0,10) arg(1,10)
sum(a,b,10)
/* maps sum to PE and hints that pointer sizes are 10. */
```

or PPC, to an instance of a processing element, for example FPGA or DSP. There are some limitations to which functions you can offload. For a generic function: *rtype f(atype par[, atype parx])*

The argument type (atype) must be a basic type (int, char, float, short) or a pointer to a basic type (int *, char*, float*, short*). For the return type (rtype), it must be a basic type only (no pointers). Currently, it is necessary to use the ”arg” directive to specify the size of the structure for pointers.

In addition, for each particular processing element there can be additional mapping restrictions:

- A function mapped to a processing element cannot invoke a function mapped to the GPP
- Global variables are not supported (if the processing element is FPGA)
- Recursive calls are not supported (if the processing element is FPGA)
- Address of stack variables are not supported (if the processing element is FPGA)

IMPORTANT: Note that the toolchain ensures feasibility with user mapping pragmas. For instance, if a mapping pragma is placed on top of a function to be mapped to a PE, all functions invoked by that function with be placed on the same PE as well.

Figure 1.2: OMP pragma parallel example

```
#pragma omp parallel
{
    {
        #pragma omp section
        functionA();
    }
    {
        #pragma omp section
        functionB();
    }
}
```

1.2 Controlling Parallelism

Bluebee toolchain adopts a subset of openMP to explicitly express parallelism. The subset is represented by the parallel and section pragmas for now. An example is given in Figure 1.2. In this example functionA and functionB may be executed in parallel (for example on independent processing elements).

The following restrictions apply in this release: Each parallel section must be enclosed in curly brackets and contain one single function call.

CHAPTER 2

Platform XML

The plaform XML describes a specific platform and backend compiler tools. This file contains the required information to allow development tools such as bbcc and bbld to invoke the necessary compilation tools for each processing element, and Harmonic to understand the capabilities of the platform. The overall organization of the XML architecture file is the following, starting from the root element:

```
<? xml version = "1.0" ?>
<PLATFORM>
  <NAME> . . . </NAME>
  <PROCESSING_ELEMENT>
    <NAME> . . . </NAME>
    <TYPE> [GPP|DSP|FPGA|GPU] </TYPE>
    <MODEL> . . . </MODEL>
    <MASTER> [YES|NO] </MASTER>
    <ENDIANESS> [LITTLE|BIG] </ENDIANESS>
    <TOOLCHAIN>
      <NAME> . . . </NAME>
        <CCOMPILER>
          <NAME> . . . </NAME>
          <VER>x.y.z</VER>
          <CMD> . . . </CMD>
          <HEADER> . . . </HEADER>
          . . .
          <HEADER> . . . </HEADER>
          <FLAGS> . . . </FLAGS>
          <BUILD_TYPE_FLAGS_MAP>
            <ALL> . . . </ALL>
            <RELEASE> . . . </RELEASE>
            <MINSIZE> . . . </MINSIZE>
            <DEBUG> . . . </DEBUG>
      <KERNDEBPROF> ... </KERNDEBPROF>
      <KERNDEBUG> ... </KERNDEBUG>
      <PROFILE> ... </PROFILE>
    </BUILD_TYPE_FLAGS_MAP>
    <OPT_FLAGS_MAP>
      <LEVEL> ... </LEVEL>
      ...
      <LEVEL> ... </LEVEL>
    </OPT_FLAGS_MAP>
    <PRECMD> ... </PRECMD>
    <POSTCMD> ... </POSTCMD>
    <COMPILE_OBJECT_RULE> ... </COMPILE_OBJECT_RULE>
    <LINK_EXECUTABLE_RULE> ... / LINK_EXECUTABLE_RULE>
    <OUTPUT_EXTENSION> ... </OUTPUT_EXTENSION>
    <ABI>
      <DATA_TYPE>
        <NAME>int</NAME>
        <PRECISION>32</PRECISION>
      </DATA_TYPE>
      ...
      <DATA_TYPE>
        <NAME>float</NAME>
        <PRECISION>32</PRECISION>
      </DATA_TYPE>
      <FUNCTION>
        <ALLOW_LEAF>[YES|NO]</ALLOW_LEAF>
        <MAX_STACK_SIZE>8192</MAX_STACK_SIZE>
        <MAX_REG_ARGS>4</MAX_REG_ARGS>
        <MAX_REG_SCRATCH>4</MAX_REG_SCRATCH>
        <CALLER_REG_SAVE>NO</CALLER_REG_SAVE>
      </FUNCTION>
    </ABI>
  </CCOMPILER>
  <LINKER>
    <NAME> ... </NAME>
    <VER>k.z.x</VER>
    <CMD> ... </CMD>
    <PRECMD> ... </PRECMD>
    <POSTCMD> ... </POSTCMD>
    <FLAGS> ... </FLAGS>
    <LINK_EXECUTABLE_RULE> ... </LINK_EXECUTABLE_RULE>
    <CREATE_STATIC_LIBRARY_RULE> ... </CREATE_STATIC_LIBRARY_RULE>
  </LINKER>
  <ARCHIVE | TOOL>
    <NAME> ... </NAME>
    <FLAGS> ... </FLAGS>
    <VER>1.2.3</VER>
    <CMD> ... </CMD>
    <PRECMD> ... </PRECMD>
```

```
     <POSTCMD> . . . </POSTCMD>
    </ARCHIVE | TOOL>
    <LIBRARY>
     <NAME> . . . </NAME>
     <VER>x.1.2</VER>
     <BUILD> . . . </BUILD>
     <LIBNAME> . . . </LIBNAME>
      . . .
     <LIBNAME> . . . </LIBNAME>
     <LIBINCLUDE> . . . </LIBINCLUDE>
      . . .
     <LIBINCLUDE> . . . </LIBINCLUDE>
     <PATH> . . . </PATH>
    </LIBRARY>
     . . .
    <LIBRARY>
     . . .
    </LIBRARY>
    <OS>
     <NAME> . . . </NAME>
     <VER>2.6.x</VER>
     <PATH> . . . </PATH>
    </OS>
   </TOOLCHAIN>
  </PROCESSING_ELEMENT>
   . . .
  <PROCESSING_ELEMENT>

  </PROCESSING_ELEMENT>
</PLATFORM>
```

2.1 PLATFORM

Platform	Type	Description
NAME	string	name of the platform
PROCESSING_ELEMENT	ProcessingElement	(multiple elements describing each functional element

ProcessingElement type

This type contains all information about the processing element required by the BB toolchain.

ProcessingElement	Type	Description
NAME	string	name of the PE
TYPE	[GPP|DSP|FPGA|GPU]	PE type
MODEL	string	PE model
MASTER	[YES|NO]	selects PE MASTER
ENDIANESS	[BIG|LITTLE]	sets the architecture endianess
TOOLCHAIN	Toolchain	describes the PE's toolchain

Toolchain type

The Toolchain type captures the development tools for each processing element, as well as features that are needed to perform a correct and efficient PE mapping. Each processing element must have available a C-compiler (*CCOMPILER*) and a linker (*LINKER*). Additional tools (*TOOL*) involved in the PE compilation can also be described.

Toolchain	Type	Description
NAME	string	name of the Toolchain
CCOMPILER	CCompiler (Tool)	describes PE C-compiler capabilities
LINKER	Linker (Tool)	describes PE linker capabilities
ARCHIVE	Archive (Tool)	describes PE archiver capabilities
TOOL	Tool	multiple elements describing additional tools
LIBRARY	Library	multiple elements describing PE libs
OS	Os	describe PE OS

Tool type

The toolchain is composed by a set of tools, and each tool is caracterized by the following items:

Tool	**Type**	**Description**
NAME	string	Tool name
VER	string	Tool version
CMD	string	Tool command string
PATH	string	Tool binary path
FLAGS	string	Tool flags
PRECMD	string	multiple commands to be executed before
POSTCMD	string	multiple Commands to be executed after
OPT_FLAGS_MAP	OptFlagsMap	mapping between bbcc/bbld optimization flags and Tool flags
BUILD_TYPE_FLAGS_MAP	BuildTypeFlagsMap	mapping between bbcc/bbld build type and Tool flags

The *bbcc* and *bbld* tools may have multiple optimization levels (-O<level>), which may be associated to a particular processing element development tool using the **OPT_FLAGS_MAP** element:

OptFlagsMap	**Type**	**Description**
LEVEL	string	multiple entries associated respectively with -O<level> (optimization level)

In addition, *bbcc* and *bbld* support multiple build configurations, each requiring a different set of flags. The **BUILD_TYPE_FLAGS_MAP** element defines this mapping:

BuildTypeFlagsMap	**Type**	**Description**
RELEASE	string	flags to be used if *RELEASE* build is selected
KERNDEBUG	string	flags to be used if *KERNDEBUG* build is selected
KERNDEBUGPROF	string	flags to be used if *KERNDEBUGPROF* build is selected
DEBUG	string	flags to be used if *DEBUG* build is selected
PROFILE	string	flags to be used if *PROFILE* build is selected
MINSIZE	string	flags to be used if *MINSIZE* build is selected
ALL	string	flags to be used if *ALL* build is selected

CCompiler type

The *CCompiler* type derives all the properties of *Tool* type, and that has the following additional fields:

CCompiler	**Type**	**Description**
HEADER	string	multiple headers to be included by hArmonic
COMPILE_OBJECT_RULE	string	command rule to compile a single object
CREATE_STATIC_LIBRARY_RULE	string	command rule to create a static library
LINK_EXECUTABLE_RULE	string	command rule to create an executable
OUTPUT_EXTENSION	string	output extension
ABI	Abi	captures PE compiler ABI features

The BB toolchain interfaces with the PE's compilers using the information stored in the *ABI* section, which describes the capabilities of each compiler:

Abi	**Type**	**Description**
DATA_TYPE	DataType	multiple sections that describe supported type and size
FUNCTION	Function	describes function ABI

The *DATA_TYPE* section is used by Harmonic to understand the supported types, the associated precision and the type cost.This information is used to derive the correct mapping and to set up transformations among processing elements:

DataType	**Type**	**String**
NAME	string	name of the C type
PRECISION	int	bit size of the type
COST	int	describes cost of the type

The *FUNCTION* section is used by hArmonic to capture compiler features related to C functions.

Function	**Type**	**Description**
ALLOW_LEAF	[YES|NO]	leaf functions are allowed?
MAX_STACK_SIZE	int	maximum stack allocation
MAX_REG_ARGS	int	number of arguments that are passed via regs
MAX_REG_SCRATCH	int	number of registers used as scratch
CALLER_REG_SAVE	[YES|NO]	is caller save?

Linker type

The Linker section describes the linker tool capabilities, including flags and build rules. The *LINKER* section supports all attributes from *TOOL*, and has the following additional fields:

Linker	**Type**	**Description**
CREATE_STATIC_LIBRARY_RULE	string	command rule to create a static library
LINK_EXECUTABLE_RULE	string	command rule to create an executable

Library type

The Library section describes PE libraries (name, include directories, build configuration). Once declared, these libraries can be referred by *NAME* in *TOOL* commands and build rules. The *LIBRARY* contains all the fields from *TOOL*, as well as the following fields:

Library	Type	Description
BUILD	BuildTypeFlagsMap	command rule to create a static library
LIBNAME	string	multiple entries of library files
LIBINCLUDE	string	multiple entries of library include directories

2.2 Platform Variables

The Platform XML file can also contain:

- Environment Variables
- Tool Variables

These variables can be expanded by BB tools (**??**), such as *bbcc* and *bbld*.

Environment Variables

The environment variables are identified by parenthesis:

```
$(HOME)
```

Environment variables are tipically used to set up PE development tool paths or to setup tool parameters. The use of variables in the platform XML file can make it more readable, for instance:

```
...
<TOOLCHAIN>
<NAME>GNU</NAME>
<CCOMPILER>
<NAME>GCC</NAME>
<VER>4.3.3</VER>
<CMD>$(TARGET_CC)</CMD>
...
```

In this example, *TARGET_CC* is an environment variable that contains the full path to the PE compiler.

Tool variables

The tool variables are indentified by curly braces:

```
${SOURCE}
```

Tool variables are set by the BB tools, and are tipically used in the *platform.xml* compiler/linker bulding rules:

```
 ...
 <COMPILE_OBJECT_RULE>
 ${COMPILER} ${CC_FLAGS} -DMASTER -D${DEFINES} -I${INCLUDE_DIRS}
-o ${OBJECT}
-I${SYSROOT}/platform/${PLATFORM}/include -I${SYSROOT}/include
-c ${SOURCE}
 </COMPILE_OBJECT_RULE>
 ...
 <LINK_EXECUTABLE_RULE>
 ${COMPILER} ${LD_FLAGS} -D${DEFINES} -I${INCLUDE_DIRS}
 ${CC_FLAGS} ${SOURCES}
-T${SYSROOT}/platform/${PLATFORM}/lib/omap3530.ld
-L${SYSROOT}/platform/${PLATFORM}/lib
-Wl,-start-group ${BBRT} ${BBRTBEAGLE}
-Wl,-end-group ${DSPLINK}
-I${SYSROOT}/platform/${PLATFORM}/include
-I${SYSROOT}/include -lpthread -L${LIBRARY_DIR} -l${LIBRARIES}
-o ${EXECUTABLE}
 </LINK_EXECUTABLE_RULE>
 ...
```

Here is the complete list of tool variables:

Variable	**Description**
ARCHIVER	archiver full path
BUILD_TYPE	active build configuration
CC_FLAGS	compiler flags inside <FLAGS> </FLAGS> of the *platform.xml*
COMPILER	compiler full path
DEFINES	list of macro in the *bbcc* command line (-D option)
EXECUTABLE	output executable *bbcc/bbld* command line (-o option)
INCLUDE_DIRS	list of include paths on the *bbcc* command line (-I option)
LD_FLAGS	linker flags inside <FLAGS> </FLAGS> of the *platform.xml*
${<LIBNAME>}	list of libraries specified inside <LIBRARY> section through <LIBNAME>
${<LIBNAME_PATH>}	list of library paths specified inside <LIBRARY> section through <LIBINCLUDE>
LIBRARIES	lists of libraries in the *bbld* command line (-l option)
LIBRARY_DIR	lists of libraries paths in the *bbld* command line (-L option)
LINKER	linker full path
OBJECT	destination object *bbcc* -o <object>
OBJECT_DIR	destination object directory *bbcc* -o <object>
OBJECTS	lists of objects in the *bbld* command line
PE	Processing element name
PLATFORM	BB platform name from *platform.xml*
SYSROOT	BB root installation path
SOURCES	list of sources on the *bbcc* command line
TMP	temporary directory used by BB tools
${<TOOLNAME>}	full path to the TOOLNAME specified in *platform.xml*
${<TOOLNAME_FLAGS>}	flags defined for TOOLNAME specified in *platform.xml*

CHAPTER 3

BBobj XML

The BBobj XML provides the format for all BB binaries. A BBobj is composed by a collection of *LIBRARY* elements. Below is a sketch of a BBobj file:

```
<?xml version="1.0"?>
<BBOBJ>
    <LIBRARY>
        <NAME>lib–name</NAME>
        <FILENAME>filename</FILENAME>
        <COMPONENT>component model | GENERIC</COMPONENT>
        <CODE>
            <DATA>
            (source OR binary in base−64 format)
            </DATA>
          <CODE\_TYPE>[C|VHDL|ASM|LIB_ELF|OBJ_ELF|EXE_ELF|BIN_COFF|PLAIN_BINARY]</CODE_TYPE>
          <TOOLCHAIN>toolchain identifier</TOOLCHAIN>
            <FLAGS>
                 <DEFINE>define C macro</DEFINE>
                   ..
                 <DEFINE>define C macro</DEFINE>
                 <INCLUDE_PATH>include directory</INCLUDE_PATH>
                   ..
                 <INCLUDE_PATH>include directory</INCLUDE_PATH>
                 <OPTIMIZATION_LEVEL>optimization level</OPTIMIZATION_LEVEL>
                 <HARMONIC>harmonic flag</HARMONIC>
                  ...
                 <HARMONIC>harmonic flag</HARMONIC>
            </FLAGS>
        </CODE>
        <VERSION>version</VERSION
        <DESCRIPTION>description</DESCRIPTION>
    <OPERATION>
      <NAME>op</NAME>
      <IMPLEMENTATION>
        <NAME>impl1</NAME>
        <HEADER> ... </HEADER>
        <HEADER> ... </HEADER>
        <ESTIMATION>
          <PERFORMANCE>val</PERFORMANCE>
          <SIZE>val</SIZE>
        <ESTIMATION>
        <PROFILED>
          <PERFORMANCE>val</PERFORMANCE>
          <SIZE>val</SIZE>
        </PROFILED>
      </IMPLEMENTATION>
      <IMPLEMENTATION> ... </IMPLEMENTATION>
    </OPERATION>
    <OPERATION> ... </OPERATION>
  </LIBRARY>
  ...
  <LIBRARY><NAME>lib name2</NAME> ... </LIBRARY>
</BBOBJ>
```

3.1 BBOBJ

BBOBJ	**Type**	**Description**
LIBRARY	Library	multiple sections describing a binary or a source

Library type

This type contains information about the supported PE library or a generic source.

Library	**Type**	**Description**
NAME	string	Library identifier
COMPONENT	string	component identifier, must match a PE model type or *GENERIC*
VERSION	string	Library version
FILENAME	string	Library file name
CODE	Code	describes and encapsulates binary or source
DESCRIPTION	string	library description
HEADER	string	multiple library include path
OPERATION	Operation	multiple description of logical function and related implementation

Code type

The Code type encodes a binary or a text in a base-64 representation, as well as information about its contents.

Code	**Type**	**Description**
CODE_TYPE	CodeType	the encoded type
TOOLCHAIN	string	Toolchain name that identifies the binary generator
FLAGS	Flag	Toolchain flags
DATA	string	encoded data

CodeType type

The code type can be one of the following:

- C - C99 source
- VHDL - VHDL source
- ASM - PE assembler
- LIB_ELF - PE ELF Library
- OBJ_ELF - PE ELF Object
- EXE_ELF - PE ELF Executable

- BIN_COFF - PE COFF binary
- PLAIN_BINARY - plain binary file

Sources may be associated to a **GENERIC** *COMPONENT*.

Flag type

The Flag type captures flags set by the *bbcc* compilation chain.

Flag	**Type**	**Description**
DEFINE	string	C macro
INCLUDE_PATH	string	multiple include paths
OPTIMIZATION_LEVEL	string	optimization level
HARMONIC	string	multiple harmonic flags

Operation type

The Operation type contains information about the functions of the library. Each function may have multiple implementations.

Operation	**Type**	**Description**
NAME	string	Operation/function name
IMPLEMENTATION	Implementation	multiple sections describing the associated implemetations

Implementation type

The Implementation type captures the estimated and profiled costs for a given implementation in order to find the best mapping.

Implementation	**Type**	**Description**
NAME	string	implementation name
ESTIMATION	Profile	estimated cost
PROFILED	Profile	real cost

Profile type

Profile	Type	Description
PERFORMANCE	int	speed based cost
SIZE	int	size based cost

CHAPTER 4

Profile XML

A BB binary, when compiled for profiling, generates profiling data in an XML format.

```
<? xml version = "1.0" ?>
<PROFILE>
 <GLOBAL>
  <MAX\_HEAP>max dynamic memory size</MAX\_HEAP>
  <APP\_TIME>application time US</APP\_TIME>
  <SYS\_TIME>remote call APIs time US</SYS\_TIME>
  <ALLOC\_TIME>memory allocation APIs time US</ALLOC\_TIME>
  <PROF\_TIME>profilation time US</PROF\_TIME>
  <SW\_CACHE\_HIT>SW cache hits</SW\_CACHE\_HIT>
  <SW\_CACHE\_MISS>SW cache miss</SW\_CACHE\_MISS>
  <PRECISION>precision scale US</PRECISION>
 </GLOBAL>
 <CALL>
  <UID>CALL UID </UID>
  <NAME>CALL name</NAME>
  <FILE>file name</FILE>
  <LINE>call line</LINE>
  <COL>call column</COL>
  <PE\_ID>PE id</PE\_ID>
  <NCALLS>number of calls</NCALLS>
  <SW\_CACHE\_HIT>hits</SW\_CACHE\_HIT>
  <SW\_CACHE\_MISS>missis</SW\_CACHE\_MISS>
  <INPUT>
   <BYTES>remote bytes transferred to PE</BYTES>
   <TIME\_TOT>total time US</TIME\_TOT>
   <TIME\_MIN>minimum time US</TIME\_MIN>
   <TIME\_MAX>maximum time US</TIME\_MAX>
  </INPUT>
  <OUTPUT>
```

```
    <BYTES>remote bytes transferred from PE</BYTES>
    <TIME\_TOT>total time US</TIME\_TOT>
    <TIME\_MIN>minimum time</TIME\_MIN>
    <TIME\_MAX>maximum time</TIME\_MAX>
  </OUTPUT>
 <IMPLEMENTATION>
    <ID>implementation ID</ID>
    <EXEC>
     <TIME\_TOT>total time US</TIME\_TOT>
     <TIME\_MIN>minimum time</TIME\_MIN>
     <TIME\_MAX>maximum time</TIME\_MAX>
    </EXEC>
    <TOT\_EXEC>
     <TIME\_TOT>total time US</TIME\_TOT>
     <TIME\_MIN>minimum time</TIME\_MIN>
     <TIME\_MAX>maximum time</TIME\_MAX>
    </TOT\_EXEC>
    <GPP\_EXEC>
     <TIME\_TOT>total time US</TIME\_TOT>
     <TIME\_MIN>minimum time</TIME\_MIN>
     <TIME\_MAX>maximum time</TIME\_MAX>
    </GPP\_EXEC>
    <GPP\_WASTE\_TIME>
     <TIME\_TOT>total time US</TIME\_TOT>
     <TIME\_MIN>minimum time</TIME\_MIN>
     <TIME\_MAX>maximum time</TIME\_MAX>
    </GPP\_WASTE\_TIME>
  </IMPLEMENTATION>
 </CALL>
</PROFILE>
```

4.1 Profile type

The profile type captures global and remote call information:

Profile	Type	Description
GLOBAL	GlobalProf	global profiling information
CALL	CallProf	multiple per call profiling information

GlobalProf type

The GlobalProf type contains global information about the profiled application:

Global	**Type**	**Description**
MAX_HEAP	int	maximum heap size in bytes
APP_TIME	int	US spent by application
PROF_TIME	int	US spent to profile
SW_CACHE_HIT	int	SW cache hit
SW_CACHE_MISS	int	SW cache miss
PRECISION	float	US maximum measuring error

CallProf type

This type contains profiling information about remote CALLs.

Call	**Type**	**Description**
UID	int	call Unique ID
NAME	string	call name
FILE	string	file that contains the call
LINE	int	line of the call
COL	int	column of the call
PE_ID	int	PE model/type
NCALLS	int	number of calls in the run
INPUT	Prof	input data call profiling information
OUTPUT	Prof	output data call profiling information
IMPLEMENTATION	Implementation	profiling information regarding the specific implementation

Prof type

The Prof type measures minimum, maximum and total time to transfer a certain number of bytes.

Prof	**Type**	**Description**
BYTES	int	C number of bytes
TIME_TOT	float	total US to transfer *BYTES*
TIME_MIN	float	minimum US to transfer *BYTES*
TIME_MAX	float	maximum US to transfer *BYTES*

Implementation type

This implementation type measures minimum, maximum and total time to execute an implementation.

Implementation	Type	Description
ID	int	Implementation ID (unique for PE model/type)
EXEC	ProfImp	remote execution time US
TOT_EXEC	ProfImp	total remote execution time US
GPP_EXEC	ProfImp	GPP execution time US
GPP_WASTE_TIME	ProfImp	GPP time spent waiting remote execution to finish

ProfImp type

ProfImp	Type	Description
TIME_TOT	float	total time US
TIME_MIN	float	minimum US
TIME_MAX	float	maximum US