Virtual Parallel Platform for
Performance Estimation

VIPPE 3.1
User Manual

April, 2020

Micro-Electronic Engineering Group
TEISA Department **University of Cantabria**
https://vippe.unican.es/

VIPPE is a complex tool & infrastructure which integrates several techniques and parts. The integration of these techniques and development of these parts has been development across several projects by the Spanish Government and the EU Commission through several research programs.

IPT-2012-0847-430000

HARP 

ART-010000-2012-5

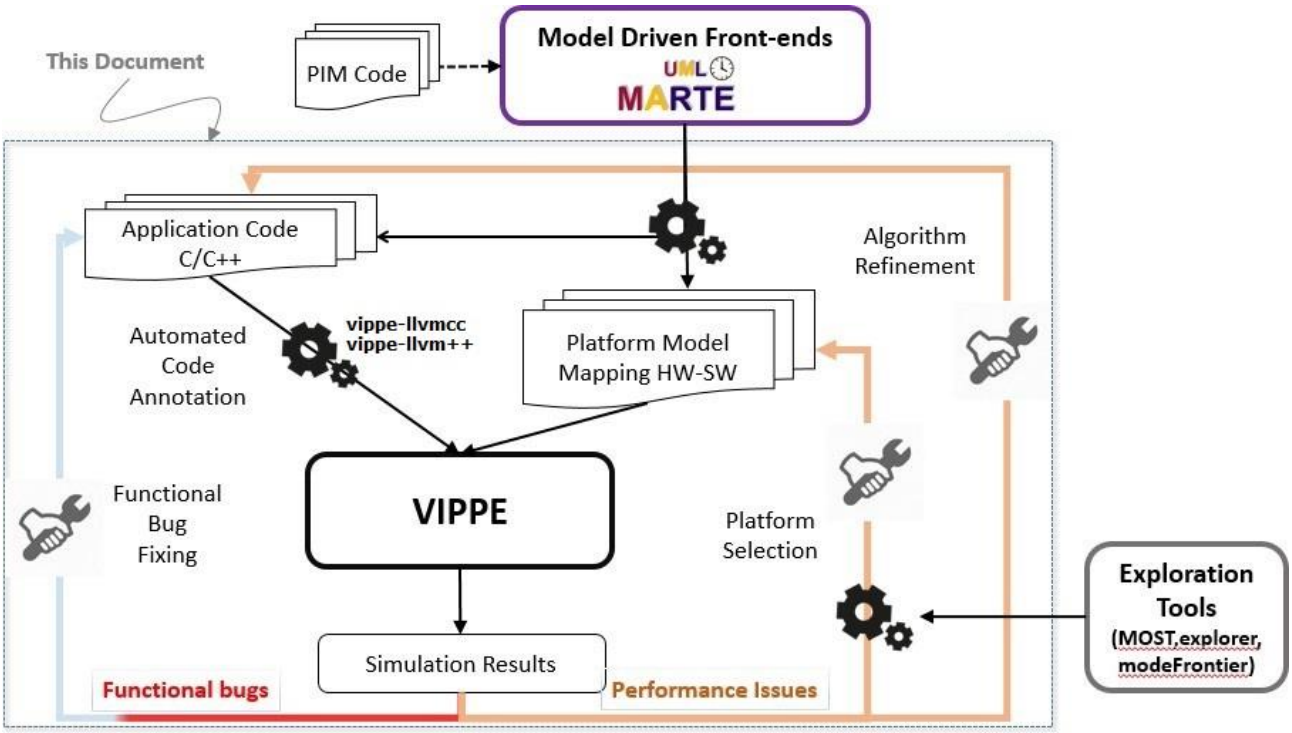
Index

1 Introduction

1.1 What's Vippe?

VIPPE is a tool infrastructure for simulation and performance estimation of complex, heterogeneous MPSoC. VIPPE allows the designer to simulate and application composed of different pieces of functionality (written in C or C++) according to a model of the target platform and of how the application functionality is mapped on such platform.

VIPPE tool has been developed in the Embedded System (GESE) group of the University of Cantabria.



VIPPE enables fast functional and performance assessment of the application targeted to a specific platform.

1.2 Release Notes

This user guide is synchronized with VIPPE version 3.1

Feature	Support	NOTES
Platform supported	Linux	Checked on Fedora and Ubuntu12.04, 14.04, and 18.4
Application Languages	C, C++	
Platform Description	C++-based, XML	
Compilation Branches	GCC, LLVM	
Supported APIs	POSIX, MPI OpenMP	
Supported ISA	arm926t, armv7, sparc microblaze	
Time Performance Estimation		Cycle Accuracy (<10% error) L1 Caches for single-core and multi-core
Energy and Power estimation	Processor	Instruction annotation level
	Bus, Peripherals	Instrumentation done
SystemC interface	With SystemC-2.3.3	Digital I/O supported interface
Exploration Interface	XML (Multicube)	

1.3 VIPPE compiler tools

VIPPE provides two tools for compiling (and automatically instrumenting) the source code under assessment.

The default VIPPE set up installs VIPPE compilers (`vippe-llvm++` and `vippe-llvmcc` tools).

As well as the VIPPE compilers it is possible to install the *source compiler*. The source compiler (`sc-vippe-g++`[1]), is an adaptation of the compiler tool distributed with SCoPE (VIPPE intercessor technology). This compiler tool has the advantage vs VIPPE compiler, that it can provide a broader support of target platforms. For instance, at the time of writing this manual, the microblaze target has been discontinued from LLVM. This means that vippe compiler does not support this target. However, you can use the source compiler as long as you have available a cross-compiler for it, e.g. mb-linux-gcc.

Moreover, the source compiler can be also the compiler of choice for your VIPPE based assessment when you assess the performance of a software whose cross-development will be performed with gcc or, in general, a non-LLVM cross compiler, as VIPPE compilers will provide the biggest accuracy when your cross development is later performed with LLVM too. The price to pay is that the source compiler is more limited than the LLVM compiler on the type of C/C++ acceptable at the input. However, the source compiler has been tested with relatively large examples with success (thousands of lines of industrial code), and it should be usable in most scenarios, as long as the code contains not so obfuscated C/C++ expressions.

[1] The source compiler uses the same executable for both C and C++ compilation. A language switch is used for precise the input source language.

1.4 Known Limitations

Feature	Issue	Expected Status
L1 data Caches modelling	Current figures are accurate only for single-core platforms	Covered
SC-VIPPE branch	Degradation of accuracy with optimizations	Under assessment
LLVM support of microblaze target	Loss of accuracy vs ECC compiler	Use the VIPPE source compiler instead standart vippe-llvm compiler
XML interface for exploration	Available but not completely Unchecked	Checked and Fixed
Energy and Power estimation	Not tuned vs any specific architecture	Covered
SystemC interface	A direct SystemC interface available. Extension to TLM2 should be manually coded right now.	Covered

2 Installation

2.1 Requirements

VIPPE has been tested to work with:

- Clang + llvm 3.5 (included in the distribution)
- 64 and32 bits operative system (Ubuntu/Fedora)
- GNU C/C++ compiler > 4.8. Versions 8, 9 and above not supported.

Gcc 7.5 recommended (included in 18.4 Ubuntu).

- LIBC version 2.15 or higher.
- LRT version 2.15 or higher.
- Gnome libxml2 (in ubuntu, packages libxml2 and libxml2-dev)
- libxml2 for 32 bits (in ubuntu, package libxml2:i386)
- Standard Linux headers and libraries for 32 bits

Ensure there is a file called libxmlc++.a or libxmlc++.so valid for 32 bits in /usr/lib... path.

Also check if there is a file called libstdc++.a or libstdc++.so valid for 32 bits in /usr/lib... path. Sometimes it is required to manually create a symbolic link. E.g. "ln -s libstdc+.so.2 libstdc++.so".

Moreover, for GUI use, you should have installed:

- libqwt-dev package
- libqt4-dev package
- qt4-dev-tools package

If you already have llvm 3.5 installed in your system, please create symbolic links to the installation and sources in *$VIPPE_HOME/utils/* folder or it will be installed again.

If you already have SystemC installed in your system, please ensure there is a *$SYSTEMC* variable in your system with its path or it will be installed again.

2.2 Installation from Sources

Untar the source distribution of VIPPE, e.g. vippe-3.1.tar.gz, e.g.:

```
$tar xzf vippe-3.1.tar.gz
```

This will create a folder with the name `$CUR_PATH/vippe-3.1` , where CUR_PATH stands for the name of the path where you untared the VIPPE distro.

Set the $`VIPPE_HOME` path:

```
$export VIPPE_HOME=$CUR_PATH/vippe-3.1
```

Once you have settled the VIPPE_HOME variable, execute the following commands on the $VIPPE_HOME folder.

```
$ make
```

This will build up the complete VIPPE infrastructure. That includes VIPPE libraries, the VIPPE asm analyzer tool, the VIPPE compiler, the VIPPE GUI and the VIPPE launchers. Moreover, after compiling the VIPPE infrastructure, the VIPPE tests will be compiled and executed.

VIPPE 3.1. works on top of LLVM 3.5 and SystemC 2.3.3. By default, make command will:

- Look if a SYSTEMC environment variable is defined. In case yes, it is interpreted that a valid SystemC set up is available, which is used for the VIPPE set up. Otherwise, the SystemC-2.3.3 libraries are installed within the $VIPPE_HOME/utile folder.
- Install LLVM-3.5 for the first time. Notice that this will require a non-negligible time. Because of that, the VIPPE *make all* and *make no_test* commands will not recompile LLVM-3.5 again. This allows recompiling VIPPE infrastructure and executing the tests much faster later times.

It is possible to generate the VIPPE infrastructure without launching tests. For that, execute the following command:

```
$ make no_tests
```

In addition, there are a complete set of makefile rules available in the Makefile placed in the root installation which allow compiling specific elements of the VIPPE infrastructure. There are also *clean*, *distclean* and *uninstall* rules for a partial or complete VIPPE remove.

Common Errors:

Error Message /usr/bin/ld: skipping incompatible /usr/lib/x86_64-linux-gnu/libxml2.a when searching for -lxml2

/usr/bin/ld: cannot find -lxml2

Besides it's installed, system can't find xml2 32bits library. We can make a symbolic link to fix this issue if we set into usr/lib/i386-linux-gnu directory and type:

Me@Me:/usr/lib/i386-linux-gnu# **ln -s libxml2.so.2 libxml2.so**

Error Message /usr/bin/ld: skipping incompatible /usr/lib/x86_64-linux-gnu/libstdc++.a when searching for –lstdc++

/usr/bin/ld: cannot find -lstdc++

Besides it's installed, system can't find stdc++ 32bits library. We can make a symbolic link to fix this issue if we set into usr/lib/i386-linux-gnu directory and type:

Me@Me:/usr/lib/i386-linux-gnu# **ln -s libstdc++.so.6 libstdc++.so**

After a successful compilation in a 64 host, you should see the following archives in the `$HOME/bin` folder:

- **vippe.x/vippe32.x**: VIPPE simulation launchers. Employed when the executable model is compiled as a set of dynamic libraries (.so). In a 64 bits host machine, vippe.x and vippe32.x shall be employed for the simulation of 32/64 bit targets. In a 32 bits host machine they are indistinguishable.

- **vippe-llvmcc/vippe-llvm++: VIPPE compilers. They are used to compile the** source code into an executable specification, either written in C or in C++ respectively.

- **asmanalyzer:** tool for the analysis of the target assembler code used in the automated instrumentation process when the "asm" estimation method is used.

This tool is invoked by the VIPPE compiler and not intended for a direct invocation from the user.

- **VIPPE*GUI tools:** Binaries part of the VIPPE GUI. These tools are invoked from the VIPPE executable model and are not intended for a direct invocation from the user.

The content of the $VIPPE_HOME/lib file should include libvippe32.a, libvippe.a, libvippeposix32.a and libvippeposix.a.

2.3 Installing the source compiler

The source compiler is not installed by default. You can install the *source compiler*, directly using the "sc_compiler" rule of the root VIPPE Makefile:

```
VIPPE_HOME$ make sc_compiler
```

This will access the $(VIPPE_HOME)/plugins/source_compiler folder and will start the compilation of the source compiler. At the end of the process you should see the links to the tools in the $(VIPPE_HOME)/bin folder.

The $(VIPPE_HOME)/bin folder shows now the `sc-vippe-g++` tool, i.e., the source compiler. The source compiler has a "-h" switch which provides a fast view of the available options and of its utilization.

2.4 Virtual Machine

The GESE Group of the University of Cantabria can provide to collaboration institutes a virtual machine with a setup of VIPPE, such installation effort is minimized. For it, please, contact the developers.

2.5 Environment

By default, the installation will involve the creation of a script file in $HOME directory of your Linux setup called *vippe_env.sh*. This file enables you to easily load all the environment variables required to compile including and linking VIPPE libraries and for invoking VIPPE tools from the command line.

The following variables need to be settled:

VIPPE_HOME: home path of VIPPE infrastructure

PATH: required to invoke VIPPE tools, e.g. vippe.x/vippe32.x, vippe-llvm++, vippe-llvmcc, VIPPEbusGUI, etc.

LD_LIBRARY_PATH: required to find VIPPE libraries, e.g. libvippe.a, lbvippe32.a, inline_functions.bc, etc.

SYSTEMC: home path for the SystemC installation

To load that environment on a recently open console:

- **`~/vippe_env.sh (or $ source ~/vippe_env.sh )`**

Finally, it is possible to install VIPPE such at the end of the installation process, the .bashrc file is directly populated with the vippe_env.sh contents. So, every time you open a console, the VIPPE environment is automatically loaded. For this, the installation of VIPPE can be performed with:

- **`Make install`**

To avoid issues running vippe_env.sh in order to add variables to system path, we can also manually edit system file called "/etc/environment", appending "/home/.../vippe-3.1/utils/llvm_install/bin:/home/…/vippe-x.x/utils/ompi_install/bin" to PATH variable.

Also append variables SYSTEMC = "/home/…/vippe-3.1/utils/systemc-2.3.3" and LIB_S3D_DIR = "/home/…/s3d_lib" if needed.

3 Basic Usage

The most basic way to use VIPPE is to compile a source .c/.cpp file (or compilation unit, that is a source file in turn including other source files) into a single executable file. Launching the VIPPE simulation consists in running such a file. The executable file is an executable and will run in the host machine. This in-host executable file allows not only check the functional behavior of the compiled software. It allows to provide accurate estimates of the time (and other extra-functional properties, i.e. energy, power) cost of each piece of code on top of the platform. For that, the code is instrumented (at the same time as compiled, for the VIPPE user) through the VIPPE compilers.

3.1 Compilation and Linking

The basic command syntax for the compilation is the following: C sources:

```
vippe-llvmcc [CFLAGS] –I$(VIPPE_HOME)/include –c SRC_FILE [-o OUTPUTNAME]
```

C++ sources:

```
vippe-llvm++ [CPPFLAGS] –I$(VIPPE_HOME)/include –c SRC_FILE [-o OUTPUTNAME]
```

By default, an object file with the same name as SRC_FILE, but substituting the .c/.cpp prefix by .o. When the –o switch is provided, the object file takes the OUTPUTNAME provided by the user.

In addition, it shall be noted that:

- Both the vippe-llvmcc and the vippe-llvm++ commands admit only a single source file (e.g. *.c or *.cpp values for SRC_FILE are not allowed).
- The shown commands use the default instrumentation method.

Once the sources are compiled, a link command of the following type will be used

```
clang++ [-m32] $(OBJS) -o $(EXE) –L$(VIPPE_HOME)/lib -L$(VIPPEPOSIX_LIB) -L$(VIPPE_LIB)
–lpthread –lrt –lxml2
```

where:

$(OBJS): refers to all the object files with the instrumented functionality produced with the VIPPE compiler

$(EXE): is the name of the executable performance model produced.

[-m32]: will be used or not in coherence with the compilation done with the VIPPE compiler. That is, it will be used if it was used with the VIPPE compiler.

$(VIPPEPOSIX_LIB): is *–lvippeposix32* if –m32 was used; *-lvippeposix* otherwise.

$(VIPPE_LIB): is *–lvippe32* if –m32 was used; *-lvippe* otherwise.

3.2 Running

For running the simulation, the user only needs to launch from the console the executable file recently created:

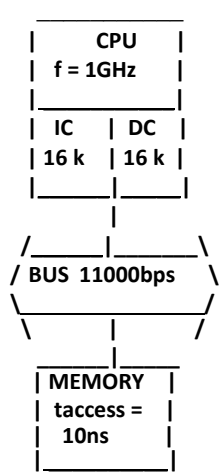
```
$./run.x
```

3.3 Console Output

For the previously shown simple example, and without further configuration, VIPPE will provide all its outputs at the console.

A first output that the user see is a message with the default platform the application is targeted to:

```
# Warning: Not XML platform definition taken, using a default platform
```



Once the end of the simulation is reached (either all user processes finish or a simulation limit is reached), then a report to the console is immediately given:

```
Simulation end 3160834524 ns
Process Elements:
    PE 0: No named
            Idle time:                  0 ns
            CPU use:                    100.000000 %
            Instructions executed:      12009019 instructions
            Instruction cache misses:   8 misses
            Data cache hits:            9002874 hits
            Data cache misses:          127 misses
            Data cache write backs:     0 writes
            Energy:     3160834560.000000 (static) + 38030051 (dynamic) = 3198864640.000000 nJ

Channel Elements:
    Channel 0: No named Accesses:   135 accesses
            Energy:     316083456.000000 (static) + 135 (dynamic) = 316083584.000000 nJ

Memory Elements: Memory 0: No named
            Accesses:   135 accesses
            Energy:     316083456.000000 (static) + 135 (dynamic) = 316083584.000000 nJ

Total system energy:        3831031808 nJ

Process 0:
            Thread 0 of process 0:
            Function:           main
            Instructions:       12009019 instructions
            Cycles:             19015029 cycles
            Data accesses:      6003003 access
            Start time:         0 ns
            End time:           3160834524 ns

            Energy:             38030050 nJ
```

As can be observed, VIPPE reports the time corresponding to the **end of the simulation**.

Following, VIPPE console report provides metric reports for three basic group of **platform elements**: Processing Elements (i.e. processors), Channel Elements (i.e. buses), and Memory Elements (i.e. memories). Notice that L1 cache-related reports given in the context of the PE report their associated to.

Finally, VIPPE console report provides **application level metrics**. The report is broken down at process-thread level. That is, metrics are provided for each thread and, in turn, for each process it belongs too.

3.4 VIPPE help

The switch "–vippe-help" will show the different options that are available for launching the simulation. This switch can be used both with the executable created through the method and with the vippe launcher application when vippe instrumented code is split into dynamic libraries.

```
$./vippe-llvmcc –vippe-help
```

--vippe-help : This help

--vippe-platform-xml <xml_platform_path> : Target platform input model to be simulated

--vippe-in [dependencies] : File inputs for VIPPE

--vippe-out [dependencies | callgrind | function_times_list | function_caches_list] : File inputs from VIPPE

--vippe-debug [platform | hw | dependencies]

--vippe-hw-bound [upper | lower] : Hardware limit to be simulated

--vippe-info : Information about compilation

--vippe-llvmui : Shows VIPPE GUIs

--vippe-systemc : Uses System-C communication infrastructure

--vippe-max-sim-time [time in seconds]: Define the max time to simulate.

--vippe-mapping-xml <folder_path>: Path of the folder containing mapping information (Mapping.xml and MemoryAllocation.xml).

--vippe-so-path <folder_path> : Path of the folder containing .so libraries.

--xml_system_configuration <filename_path> : (Multicube) system configuration file (input file fixes the values of the parameters in a VIPPE configurable model).

--xml_system_metrics <filename_path>: (Multicube) system metrics file (name of the output file where VIPPE will dump the assessed metrics).

The next table provides a more detailed explanation of all these switches:

Vippe Switch Explanation
--vippe-help
This Help
--vippe-platform-xml <xml_platform_path>
To provide the folder where the description of the target platform is
--vippe-in [dependencies]
File inputs for VIPPE dependency analysis
--vippe-out [deps | callgrind | function_times_list | function_caches_list | response_times]
Provides information on dependencies, or a callgrind compatible report, or a sequences of calls report (with start-end times) for each thread or process, or information of cache performance for each function, or response times for each function report (min., max., average and # calls).
--vippe-debug [platform | hw | dependencies]
TBC
--vippe-hw-bound [upper | lower]
Hardware (time) performance limit to be simulated
--vippe-info
Information about compilation
--vippe-gui
Shows VIPPE GUI along simulation
--vippe-systemc
Uses SystemC communication infrastructure
--vippe-max-sim-time [time in seconds]
Define the max time to simulate
--vippe-mapping-xml <folder_path>
Path of the folder containing mapping information, e.g. of memory spaces to OS instances, of OS instances to processing elements (Mapping.xml and MemoryAllocation.xml).
--vippe-so-path <folder_path>
Path of the folder containing dynamic libraries (.so) containing the instrumented code.
--xml_system_configuration <filename_path>
Multicube system configuration file. This is an input file to a VIPPE configurable model. A VIPPE configurable model has configurable values which have to be set before the simulation starts. The system configuration file fixes the values of those parameters.
--xml_system_metrics <filename_path>
Multicube system metrics file. Output file name where VIPPE dump the assessed metrics.

4 Compiling the application

The VIPPE compiler provides its own compiling infrastructure for automatically instrument the source code. That instrumentation is later used by the VIPPE simulation and back-end infrastructure to provide the performance figures it is capable to report. As mentioned in section 3.1.1, VIPPE framework enables two compilation tools. VIPPE main compiler branch is the vippe compiler (`vippe-llvmcc` and `vippe-llvmcc`) and the sc-compiler (and adaptation of the ancestor SCoPE tool). Using the VIPPE compiler method provides optimum accuracy results when the cross-development is going to rely on LLVM compiler too. Using the sc compiler is suitable for better accuracy when the cross-development will not rely on LLVM but, for instance, on gcc[2].

4.1 VIPPE compiler

4.1.1 Compilation Process

VIPPE simulator uses a compiler based on LLVM (for further details http://llvm.org/) to instrument the user code adding required marks for simulating.

VIPPE compiler is based on LLVM compiler (Clang). LLVM enables to create optimization steps. During compilation process VIPPE instruments the user code by this optimization steps.

When compiling a user code, the instrumentation process is applied to each source file and follows the next steps:

- First step: The compiler (Clang) translates the user code to an intermediate code (LLVM compatible).
- Second step: The code obtained in the first step is parsed using LLVM optimization pass to instrument it with the annotations for the simulator required metrics.
- Third step: Then, the compiler (Clang) applies the rest of the optimization passes. It means compiler flags such as O2 or whatever supported by the compiler.
- Fourth step: The file from the previous step is compiled and a binary object is obtained.

Apart from the supported LLVM compiler (Clang) options, VIPPE compiler supports more options to enable the optimization steps.

[2] Limitations on the admitted grammar can be found. Moreover, some features might not be available currently when using the sc compiler, e.g. response times report

Thus, VIPPE compiler supports the following options:

OPTION	DESCRIPTION
--vippe-cpu=CPU_NAME	Name of the cpu where the code is expected to work. For estimation purposes
--vippe-verbose	Prints intermediate parsing commands
--vippe-preserve-files	Preserves intermediate parsing files
--vippe-nocaches	Disables annotating cache marks
--vippe-coverage	Enables source code coverage measures
--vippe-notimes	Disables time and cache annotations
--vippe-fannotate	Enables time function annotations
--vippe-ftimes[=file]	Enable time function annotations
--vippe-fcaches[=file]	Enable cache accesses function annotations
--vippe-dependencies = file.xml	Enables taking semaphore dependencies among threads (RD, RD_WR, and WR)
--vippe-asm	ASM Analysis
--help	Prints vippe-llvm++ / vippe-llvmcc help

Figure 1 - Table of VIPPE compiler supported options

In order to model 32 bits systems, include the -m32 flag during compilation.

VIPPE compiler supports three ways to compile the source code to be estimated.

4.1.2 LLVM IR based Annotation (Default)

This is the default compilation mechanisms for VIPPE, therefore, the same command syntax shown in section 5.1.1 can be used.

This way the instrumentation will automatically account one instruction (and cycle) per instruction of the LLVM internal representation (IR). Since the LLVM IR does not matches a specific target ISA, this will provide an approximate idea of the computational load of each piece of code. The advantage of this compilation method is that is as portable as LLVM itself and can serve for a first, prospective study.

4.1.3 ASM Annotation method

This is the most accurate method for assessing the performance of the source code. VIPPE will annotate at the LLVM IR level. However, VIPPE will not associate computational loads assessed from this intermediate representation, but it will associate computational loads extracted and estimated from the actual binary code which would be produced for the actual target through the LLVM back-end.

The command for compiling with the ASM annotation method is as follows:

C sources:

```
vippe-llvmcc -m32 [CFLAGS] –vippe-asm –vippe-cpu=TARGET_ISA -I$(VIPPE_HOME)/include -c [-o OUTPUTNAME]
```

C++ sources:

```
vippe-llvm++ -m32 [CPPLFAGS] –vippe-asm –vippe-cpu=TARGET_ISA -I$(VIPPE_HOME)/include -c [-o OUTPUTNAME]
```

where TARGET_ISA refers to the target instruction set architecture. The targets set can be extended as explained forward.

In order to obtain the highest accuracy, it is also required to consider the target width.

Assuming the development is done in a64bits platform, the assessment of 32bits targets requires to add the –m32 switch. That is:

C sources:

```
vippe-llvmcc -m32 [CLFAGS] –vippe-asm –vippe-cpu=TARGET_ISA -I$(VIPPE_HOME)/include -c [-o OUTPUTNAME]
```

C++ sources:

```
vippe-llvm++ -m32 [CPPLFAGS] –vippe-asm –vippe-cpu=TARGET_ISA -I$(VIPPE_HOME)/include -c [-o OUTPUTNAME]
```

If the –m32 flag is not used, the annotation is done for 64bit targets.

NOTES:

- The –m32 switch must be coherent with the selected target. Currently, no consistency check is done in that sense.
- Currently, merging of instrumentations for simulating mappings of the software to platforms heterogeneous in ISA is supported.
- The support of platforms heterogeneous in bit width has not be assessed so far.
- When compiling with this method, the user could see warning messages at compilation time of the type “Unknown code OPCODE_NAME”. This means that when the binary analysis is done, the operation code OPCODE_NAME has not been recognized. This will mean in general some loss of accuracy, but does not prevent the compilation.

4.1.4 No Annotation

In the compilation of a VIPPE executable model, one or more source files can be compiled without annotation. For compiling one source file without annotations, just use the **`-vippe-notimes`** switch.

C sources:

```
vippe-llvmcc –m32 [CFLAGS] --vippe-notimes –I$(VIPPE_HOME)/include –c
[-o OUTPUTNAME]
```

C++ sources:

```
vippe-llvm++ -m32 [CPPFLAGS] --vippe-notimes –I$(VIPPE_HOME)/include
–c [-o OUTPUTNAME]
```

The different types of instrumentations can be merged in a compilation of an application split into several compilation units.

An example is shown in the *$VIPPE_HOME/accuracy/multifile*.

4.2 Source Compiler in VIPPE

As was mentioned, the source compiler is applicable for estimating targets not supported by LLVM and if the user will not perform the cross-development with LLVM. The basic command syntax is the following:

- **`sc-vippe-g++ [options] [gcc-options] --sc-method=METHOD input-file`**

The command has a help switch:

- **`sc-vippe-g++ -h`**

Under this syntax, [options] refers to all the options specific to the VIPPE source compiler. [gcc-options] refer to options which would be passed to a gcc compiler.

As can be seen, there are two mandatory parameters:

input-file: The source file to be compiled

--sc-method=METHOD: The SW estimation method, which determines the way to instrument. METHOD can adopt one among the following values: *asm-sentences*, *asm- opcodes*, *none*.

Following subsections exemplify the compilation commands for the *bubble_sc_comp* example (in *$(VIPPE_HOME)/accuracy/bubble_sc_comp*).

In all these cases, the later linkage is done as it is when the VIPPE LLVM-based compiler is used.

4.2.1 Operation code-based instrumentation

An example of compilation command for the operation code-based instrumentation is given as follows:

In this example notice that:

- The sc_language option serves for stating the language of the source file. It can be either 'c' or 'c++'. In the former case, the name of the executable "sc- vippe-g++" remains the same. That is, for compiling a C source file, the command `sc-vippe-g++ --sc-language='c'` shall be used.
- The `-sc-m32` option servers for targeting the estimation of a 32bits target.
- The `-sc-crosscompiler` option servers to provide the crosscompiler tool name used in the binary-based analysis[3]. By default, the sc-vippe compiler will use a linux compiler[4], and specifically and ARM linux compiler (arm-linux-gcc). However, the user can use this option to specific the exact command name. For instance, in the exemplified command, the available ARM linux compiler has a slightly different prefix. The user can either, perform a soft link, or as shown, provide the exact compiler name to the sc-vippe compiler command.

  ➢ **sc-vippe-g++ [-O2] --sc-language=c++ --sc-m32 --sc-crosscompiler=arm-linux-gnueabi-g++ --sc-cpu=armv7a --sc-method=asm-opcodes -c bubble.cpp -o bubble.o**

- `--sc-cpu` option is used for providing the exact target ISA architecture. The same target ISAs as for the VIPPE LLVM-based compiler are supported.
- The `-c` flag is required for compiling olny (without linking) with the SC compiler

4.2.2 ASM sentences annotation

For compiling the bubble_sc_comp example through the ASM sentences analysis, the following command will be used:

➢ **sc-vippe-g++ [-O2] --sc-language=c++ --sc-m32 --sc-crosscompiler=arm-linux-gnueabi-g++ --sc-cpu=armv7a --sc-method=asm-sentences -c bubble.cpp -o bubble.o**

4.2.3 No annotation

For compiling the *bubble_sc_comp* example without no type of performance (time, or cache annotations) annotation:

➢ **sc-vippe-g++ [-O2] --sc-language=c++ --sc-m32 --sc-noannotate -c bubble.cpp -o bubble.o**

This way, only RTOS related transformation will be performed.

[3] The SC compiler relies on the availability of the cross compiler for the binary-based analysis.

[4] The linux-based compiler is used even if the estimation is done for a bare-metal application. The reason is related to the current way VIPPE covers the estimation of _DIR based applications.

4.3 Setting TARGET ISA and extending targeting capabilities

The most accurate instrumentation methods of VIPPE rely on the association of performance costs to the instructions of the target ISA. VIPPE distribution supports by default a number of target ISA. Currently, any of the following values are possible:

```
TARGET_ISA = arm926t | armv7a | sparc | microblaze
```

The user can extend the targeting capabilities in case none of the above TARGET ISA are the ones that the user wants to target.

For this extension, two files, placed in the *$VIPPE_HOME/config* folder of the VIPPE distribution have to be extended, the *opcode_cost.xml* file and */meminst.xml* file.

The *opcode_cost.xml* contains the performance costs associated to each instruction for every supported TARGET_ISA. Figure 2 shows its basic structure.

```
<processors>
        <processor type="arm926t" >
              <costs>
                    <cost opcode="clz" cycles="1" energy_pj="2" />
                    <cost opcode="adr" cycles="1" energy_pj="2" />
                    <cost opcode="and" cycles="1" energy_pj="2" />
                     ...
                    <cost opcode="bne" cycles="1" energy_pj="2" />
                    <cost opcode="default" cycles="1" energy_pj="2" />
              </costs>
         </processor>
        ...
        <processor type="newisa" >
              <costs>
                    <!-- Add here the power PC instr. costs-->
                    <!-  <cost opcode="" cycles="" energy_pj="" />
                    ... -->
                    <cost opcode="default" cycles="1" energy_pj="2" />
              </costs>
         </processor>
</processors>
```

Figure 2 - Performance costs configuration and adding a minimal description for a new target ISA.

There is a “processor” tag per target ISA. Within the “processor” tag, the “costs” tags limit the “costs” entries for each instruction. The performance costs of each instruction are settled with the “cost” tag. For each “cost” entry the user settles the operation code of the instruction (i.e. the assembly name of the instruction), and two further entries, “cycles” for the amount of cycles per instruction, and “energy” for the amount of energy consumed by the instruction[5].

The user does not need to provide a complete list of costs for all the instructions of the target, e.g. if there is no sufficient information on it. The user can use the “default” value for the “opcode” attribute. This will mean that for every operation code not recognized during at the analysis performed by the VIPPE compiler (reported with warning messages of the type “Unknown opcode”), the default costs will be assigned at simulation time.

Figure 2 also illustrates a minimal definition of a target ISA called “newisa”. In such a minimal definition, the default cost is stating the same costs in cycles and instructions for every executed instruction (regardless its type).

The user can use this configuration file for maximum and minimum time estimations. For instance, it is possible to add a “newisa” and a “newisa_max” target ISAs and use them at compilation times with the corresponding “—vippe-cpu” or “—sc-cpu” options, for the VIPPE LLVM-based compiler and VIPPE source compiler respectively.

As has been mentioned, for completing the extension the user needs to edit also the *meminst.xml* file, which has the format shown in Figure 3. The meminst.xmls file is used to declare the load, store and jump instructions for each supported target ISA. This information is used in data cache performance estimation.

Figure 3 also illustrates the minimal extension required. At least one operational code of load type and one of store type (even if the operation codes are fake ones) is required.

[5] VIPPE does not perform a micro-architectural modelling, i.e. modelling of the pipe. Indeed, this type of modelling is very time consuming as it can easily lead to orders-of-magnitude of simulation speed degradation. An approach to approximate more to that modelling is to adjust the costs according to a regression test, e.g. instead of using the nominal cycles of a datasheet.

```
<processors>
  <processor type="armv7a" >
      <loads>
          <load opcode="ldr" />
          <load opcode="ldrh" />
          <load opcode="ldrb" />
          <load opcode="ldrsh" />
          <load opcode="ldrsb" />
          <load opcode="ldm" />
          <load opcode="ldmia" />
          <load opcode="ldmfd" />
      </loads>
      <stores>
          <store opcode="str" />
          <store opcode="strb" />
          <store opcode="strh" />
          <store opcode="strsh" />
          <store opcode="strsb" />
          <store opcode="stm" />
          <store opcode="stmia" />
          <store opcode="stmfd" />
      </stores>
      <jumps>
          <jump opcode="b" />
      </jumps>
  </processor>
...
  <processor type="newisa" >
      <loads>
          <load opcode="ld" />
      </loads>
      <stores>
          <store opcode="sd" />
      </stores>
  </processor>
</processors>
```

Figure 3 - The *meminst.xml* file captures the store, load and jump instructions for each supported target.

5 Modelling Heterogeneous Systems

VIPPE allows for the modelling and simulation of heterogeneous systems in terms of ISA. Specifically, as was advanced in section 6.1.2, VIPPE can simulate cases where some functionality (enclosed in one or more compilation units) is instrumented modelling that will be run in one type of processor, while other functionality (enclosed in one or more compilation units) is instrumented to model that it will be run in a different type of processor.

In order to do that it is required to introduce first how VIPPE infrastructure supports separated compilation and the generation of an executable model as a set of dynamic libraries, and to introduce for the latter case the VIPPE launcher

5.1 Separated Compilation

VIPPE supports separate compilation. That is, the functionality can be separated into several source files and compiled into several compilation units. Each compilation unit is a .o file. One source compilation file will correspond to one compilation unit. That source file can include an arbitrary number of source files.

The *$VIPPE_HOME/examples/multifile/* folder contains an example of separated compilation. For instance, when using the "notimes" rule of the Makefile (`make notimes`) on that example, the following sequence of commands is invoked:

```
make notimes
vippe-llvm++ --vippe-notimes main.cpp -o main_notimes.o -m32
vippe-llvm++ --vippe-notimes file2.cpp -o file2_notimes.o -m32
clang++ -m32 main_notimes.o file2_notimes.o -o run_notimes.x -I$VIPPE_HOME/include –
      L$VIPPE_HOME/lib -lvippeposix32 -lvippe32 -lpthread -lrt -lxml2
```

This will result in an executable model called "run_notimes.x". In this example, all the compilation units have been compiled with the same type of annotation (actually, no annotation). Therefore, the execution of this model will produce no time accounting, serving just for functional validation.

VIPPE allows to merge different types of annotation. In the same example, it is possible to execute also the "mix" rule. The following sequence of compilation commands will be invoked:

```
make mix
vippe-llvm++ --vippe-notimes main.cpp -o main_notimes.o -m32
vippe-llvm++ --vippe-asm --vippe-cpu=armv7a file2.cpp -o file2_asm.o -m32
clang++ -m32 main_notimes.o file2_asm.o -o run_mix.x -I$VIPPE_HOME/include –
  L$VIPPE_HOME/lib -lvippeposix32 -lvippe32 -lpthread -lrt -lxml2
```

This sequence ends up producing the "run_mix.x" executable file, where only the "file2.cpp" functionality is annotated and will have relevance to the effects of performance accounting.

5.2 Compiling functionality into dynamic libraries and VIPPE launcher

5.2.1 Compiling functionality into dynamic libraries

In the examples shown so far, the user produces an executable model out of the sources of the user application(s). That executable specification can be run directly, which models its execution on top of the "VIPPE default platform". Additional switches can be added to the run command (--vippe-platform-xml and –vippe-mapping-xml) for stating the platform and how the application(s) is(are) mapped to the platform. It has also seen (previous section) that it is possible to compile source functionality in different static compilation units.

In addition, in VIPPE is also possible to compile the functionality into a set of one or more dynamic libraries. Each of those dynamic libraries can have a different type of annotation (notimes, IR, ASM). This way, for instance, some dynamic libraries can have annotated time, while not others. The compilation of the functionality into dynamic libraries facilitates also the modeling of heterogeneous platforms, as different dynamic libraries can use different instrumentation to refer to different targets[6].

This capability for encapsulating the functionality into dynamic libraries is indeed exploited by the CONTREX Eclipse plug-in [16], which from a component-based application model in UML/MARTE, to map "memory spaces" (a specific UML/MARTE element in the UML/MARTE methodology [15]) into these dynamic libraries.

The *$VIPPE_HOME/examples/bubble_so/* folder contains a simple example where the bubble functionality is compiled into a single dynamic library. By typing "make", the following sequence of commands is generated for the compilation of the dynamic library:

- **vippe-llvm++ --vippe-verbose --vippe-preserve-files --vippe-asm --vippe-cpu=armv7a bubble.cpp -o bubble.o - m32**
- **g++ -m32 -shared -fPIC -o memoryspace.so bubble.o -L$VIPPE_HOME/lib -lpthread -lrt - lxml2**

This sequence of commands generates the dynamic library, a file called "memoryspace.so", where a single object file, bubble.o is included. More object files with the instrumented functionality can be added to the dynamic library by generating additional object files and adding them at the linking command right after "bubble.o".

[6] Currently, all the dynamic libraries should be compiled either for 32bits or 64bits. This is a current limitation to the heterogeneity of the modelled platform.

5.2.2 The VIPPE launcher

Once the functionality has been compiled into a set of dynamic libraries, it is required to launch the execution of them.

For that, VIPPE provides the VIPPE launcher tool. There are two versions, "vippe.x" for launching and estimating 64bit target platforms (which can be run only in 64bits hosts), and "vipp32.x" for launching and estimating32bits target platforms.

A basic option of the vippe launcher is to launch its help message to console:

```
$ vippe32.x --vippe-help
--vippe-help : This help
--vippe-platform-xml <xml_platform_path> : Target platform input model to be simulated
--vippe-in [dependencies] : File inputs for VIPPE
--vippe-out [dependencies | callgrind | function_times_list | function_caches_list | response_times]
      : File inputs from VIPPE
--vippe-debug [platform | hw | dependencies]
--vippe-hw-bound [upper | lower] : Hardware limit to be simulated
--vippe-info : Information about compilation
--vippe-gui : Shows VIPPE GUIs
--vippe-systemc : Uses System-C communication infrastructure
--vippe-max-sim-time [time in seconds]: Define the max time to simulate.
--vippe-mapping-xml <folder_path> : Path of the folder containing mapping information
      (Mapping.xml and MemoryAllocation.xml).
--vippe-so-path <folder_path> : Path of the folder containing .so libraries.
--xml_system_configuration <filename_path> : (Multicube) system configuration file (input file
      fixes the values of the parameters in a VIPPE configurable model).
--xml_system_metrics <filename_path> : (Multicube) system metrics file (name of the output file
      where VIPPE will dump the assessed metrics).
--xml_metric_definition <filename_path> : Input file where VIPPE reads which are the performance
      metrics to report in the system metrics file.
```

As can be observed, the same options as when an executable file is created are available. Specifically, it is possible to provide, through the "vippe-platform-xml" an XML description of the platform.

Moreover, the VIPPE launcher has the "vippe-so-path" switch. This switch enables the specification of a folder where all the dynamic libraries encapsulating the functionality to be simulated are place.

Thus, for instance, in order to run the example in *$VIPPE_HOME/examples/bubble_so/*, the following command is generated (produced with "make run"):

- **vippe32.x --vippe-platform-xml xmlfiles/ --vippe-mapping-xml xmlfiles/ --vippe-so-path ./**

Thus, for instance, in this case, the parameter passed to the "—vippe-so-path" switch is "./", since the example Makefiles leaves the "memoryspace.so" library right in ".".

The VIPPE launcher parses the **"Mapping.xml"** file for deciding which dynamic library has to be run on which OS instance7.

The following code excerpt shows the "*Mapping.xml"* file for the example in *$VIPPE_HOME/examples/bubble_so/*:

```
<?xml version='1.0' encoding='UTF-8'?>
      <Mapping_Allocation>
            <SW_allocation>
                <SW_instance name="rtos1">
                    <memory_partition name="memoryspace"/>
                    <HW_instance name="cpu1"/>
                    <HW_instance name="cpu2"/>
                </SW_instance>
            </SW_allocation>
            ...
      </Mapping_Allocation>
```

The information is within the "SW_allocation". This section contains a number of RTOS instance entries previously declared in the SW platform description, in the "SWPlatform.xml" file (see Section 8). Each RTOS instance is identified by the "SW instance" tag and its instance name, e.g. "rtos1" in the example. For each RTOS instance each dynamic library mapped (running on) the RTOS is stated trough the "memory_partition" tag. The "name" attribute of the memory partition tag states the "base name" of the dynamic library. That is, the dynamic library running on the "rtos1" OS in the example will be called base_name+".so", that is, "memoryspace.so".

5.3 Communication across RTOS

There are two possibilities for communicating code mapped to two different RTOS instances:

- Across a shared memory directly mappable at both RTOS memory maps
- Across a network

5.3.1 Shared Memory across-RTOS communication

VIPPE allows to model an efficient communication mechanism, shared memory, on the following situations

- Two applications which run on top of different RTOS and can directly access a shared memory accessible in their memory maps.
- Two bare metal applications running on top of different processors which see a shared memory which is directly accessible on both memory maps.
- A combined case, one bare metal application on one side of the shared a memory and one application on top of a RTOS allowing a directly mapped access to the shared memory.

For simplicity, VIPPE assumes the mapping of this shared memory being based on the same offset,0x0, on both sides. The size of this shared memory is fixed (16MB by default, and can be changed at VIPPE compilation). At compilation time, VIPPE enables to not to include this memory model.

For using this type of communication, the user code needs to include the VIPPE *Uc_phy_sh_mem.h* header file. This header declares the following access functions:

```
int uc_phy_write(void *addr, void *ptr, int len);
int uc_phy_read(void *addr, void *ptr, int len);
```

These functions allow to write/read a bunch of data to/from the modelled directly mapped memory. This way, for instance, two applications running in different RTOS mapped to different processors communicated via a shared memory (at the platform) can be modelled as in the following example. The application writing to the memory, eg. in RTO1 mapped in PE1, can define a function as the one defined in the excerpt of code in Figure 4.

[7] This file also contains the information which states the OS to processing resources mapping (see section 8).

```
#include "uc_phy_sh_mem.h"

void sh_write(int var) {
     printf("Executing sh_write\n");
     void *addr=0x12345;
     int len=4;
      uc_phy_write(addr, &var, len);
      return;
}
```

Figure 4 - An excerpt of code writing to a directly mapped shared memory.

The application reading from the memory, e.g. in RTOS2 mapped to PE2, can define a function as the one defined in the excerpt of code in Figure 5:

```
#include "uc_phy_sh_mem.h"

void sh_read(int *var) {
        printf("\t\tExecuting sh_read\n");
        void *addr = 0x12345;
        int len = 4;
        uc_phy_read (addr, var, len);
        return;
}
```

Figure 5 - An excerpt of user code reading from a directly mapped shared memory.

The shared memory is assumed to be located between PE1 and PE2 (no explicitly captured in the XML front-end), and be mapped for the two PE buses in the same base address (0x0). In the example the address0x12345 is used over the based address, and, by default, up to16 MB are available for that communication.

6 Describing the Platform and the Mapping

6.1 Attribute-based default units and default attribute values

For user commodity, VIPPE user default units and default parameters.

Default units means that the user can omit in the description of the attribute units. In that case, a default unit is assumed. The default unit depends on the specific attribute (not on the physical magnitude). For instance, for a main memory, VIPPE XML front- end will assume the user refers to MB if a memory size value has not unit. However, for a bus wordwidth, the default unit assumed is the byte ("byte" or "B").

VIPPE supports also default values for specific attributes in the description of a platform. For instance, a user can instance a main memory and not providing a specific size value. In that case, VIPPE assumes a size of500MB. This will involve a warning report in the console while VIPPE performs the XML parsing phase.

The following table reports the default units and default values for attributes in VIPPE:

Component	Attribute	Default unit	Default Value
Processor	"frequency"	MHz	1GHz
	"static_power"	W,watt	1W
	"cycle_consumption"	nJ	1nJ
Instructions & Data L1 Cache	"mem_size"	B,byte	16KB
	"wordSize"	B,byte	Wordwidth of bus attached to processor containing the cache memory
	"blockSize"	W, word	8W (W depends on the wordsize)

	“associativity”	-	1
	“writePolicy”	-	“writeBack”
	“static_power”	W,watt	1W
	“hit_consumption”	nJ	1nJ
	“miss_consumption”	nJ	1nJ
Memory	“mem_size”	MB, MByte	500MB
	“mem_latency”	ns	1ns
	“static_power”	W,watt	1W
	“access_consumption”	nJ	1nJ
Bus (channel)	“width”	B, byte	
	“bandWidth”	b/s,bps	1000000bps
	“burstSize”	W, word	8W
	“static_power”	W,watt	1W
	“access_consumption”	nJ	1nJ
Reconf. HW	Internal memory size “mem_size”	MB, MByte	0MB
	Internal memory latency “mem_latency”	ns	0ns

7 Outputs

Section 3.3 provided a first flavor of the output capabilities of VIPPE. Following sections provide a more detailed view of the metrics and ways to obtain those reports.

7.1 Console report

Following example illustrates in detail the output report.

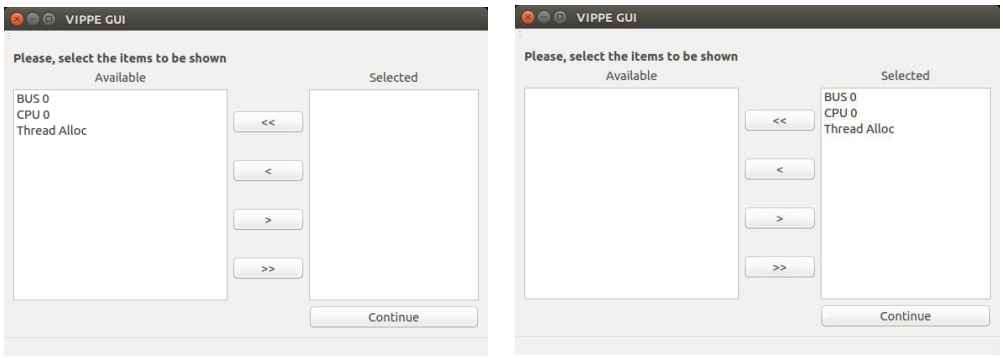
7.2 Graphic output

VIPPE can be launched to provide a graphical output. For that, you need to launch the vippe simulation with the `-vippe-gui` switch[8].

For example, after compiling the *bubble_gui* example in *$(VIPPE_HOME)/accuracy/bubble_gui* the run.x executable model is generated. In order to launch the model with the GUI you execute:

```
$./run.x -vippe-gui
```

This shows a GUI selector window as the one shown in Figure 6. In that window, the user can select among the available graphical reports. The selector window will provide choices after recognizing the processor and bus instances present in the executable model. In the Figure 6 case, there is only one processor and one bus report related window. In addition, the user can also select a "ThreadAlloc" window. All these **graphical reports are dynamic** and evolve together with the simulation time. They will freeze once the simulation is finished.



(a) (b)

Figure 6 - VIPPE GUI selection window once the simulation is started (a) and right after selecting all the GUIs available (b).

[8] For this, VIPPE has had to be compiled with the required dependencies for GUI (Qt4).

Figure 7 shows the graphical report associated to the processing element. The user can check and enable the CPU utilization (“CPU use” check box), the amount of executed instructions (“exec. Instr” check button) and the amount of bus accesses (“Buss access” check button). The latter two are reported in the same (button) graph if both are selected.

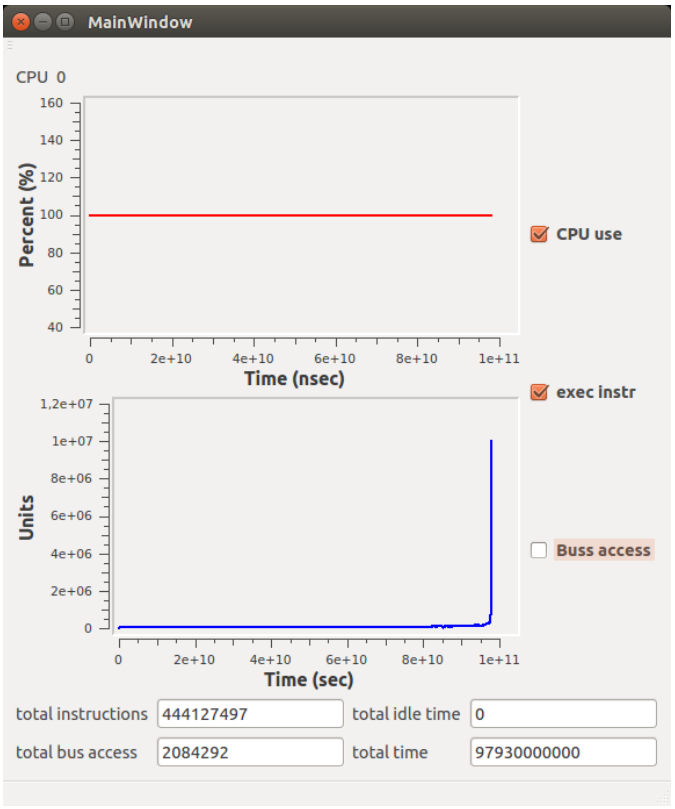


Figure 7 - Processing element graphical report window.

Figure 8 shows the bus graphical report, which shows the bus congestion.

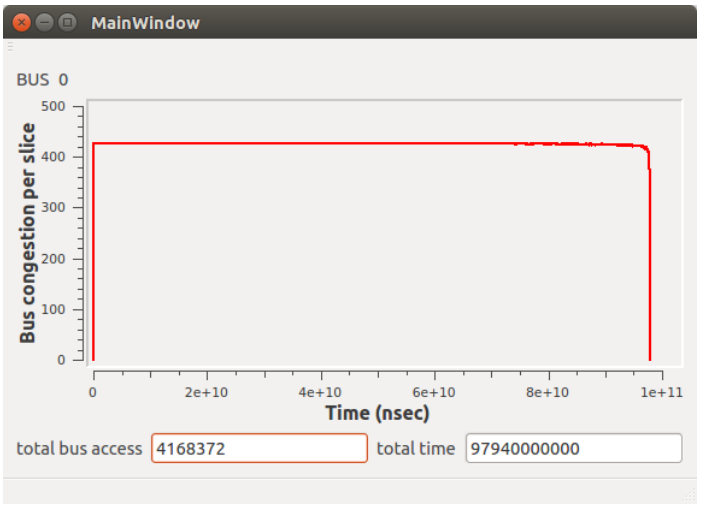


Figure 8 -Communication element graphical report window.

Figure 9 shows the gannt chart shown with the "ThreadAlloc" graphical report. It shows how each thread is scheduled on each processing core across time. A color code is assigned to each thread.

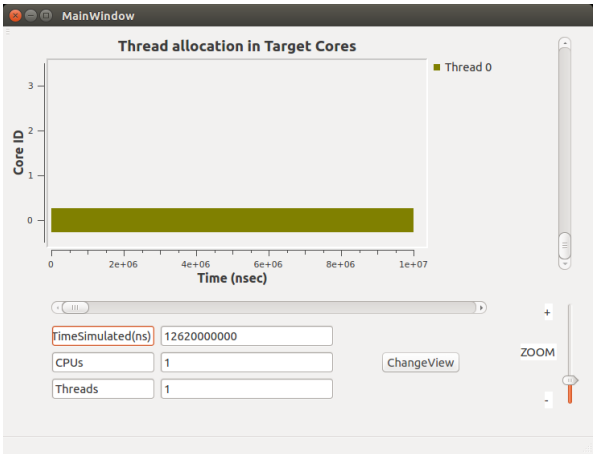


Figure 9 - "ThreadAlloc" graphical report window.

7.3 XML output

VIPPE can provide performance metrics on an XML format compliant with the XML Multicube specification.

By default, VIPPE generates the XML output file with the name output.xml. The user can change the name via the following command:

7.4 Callgrind and Response Times

VIPPE can provide detailed metrics on the response times. This is explained in section 10.

7.5 Power Traces

VIPPE can provide power traces, which report dynamic and static power consumption per each processor instance of the system model along time. The output format is compatible with the input reads by ThermalProfiler from Intel Docea, which enables dynamic thermal analysis.

For enabling the export of the traces, the user shall add and additional `--vippe-out power_traces` switch as illustrated below:

- **vippe32.x --vippe-platform-xml xmlfiles/ --vippe-mapping-xml xmlfiles/ --vippe-so-path ./ --vippe-out power_traces**

A single file in VCD (Value Change Dump) format is generated. The fastest update rate of the power values is the VIPPE time slice (10ms by default), as this is the time when VIPPE calculates the (dynamic and static) energy consumptions for each processing element) for the *primitive* metrics internally accounted by VIPPE (e.g., instructions executed, cache misses, etc). Then, the energy calculated is divided by the time slice.

The name of the power traces file generated takes the format *"power_traces_$date_$time.vcd"*. Therefore, it is possible to perform several simulations of the systems without collision of the power trace reports.

```
$comment
      Power trace file generated by VIPPE tool
$end

$timescale 1ms
$end

$scope module VIPPE/ARM1 $end
      $var real 1 a leak_power_ARM1 $end
      $var real 1 b dyn_power_ARM1 $end
$upscope $end

$scope module VIPPE/ARM2 $end
      $var real 1 c leak_power_ARM2 $end
      $var real 1 d dyn_power_ARM2 $end
$upscope $end

$scope module VIPPE/MB1 $end
      $var real 1 e leak_power_MB1 $end
      $var real 1 f dyn_power_MB1 $end
$upscope $end

$scope module VIPPE/MB2 $end
      $var real 1 g leak_power_MB2 $end
      $var real 1 h dyn_power_MB2 $end
$upscope $end

$enddefinitions $end


#0 r0.000000 a r0.000000 b r0.000000 c r0.000000 d r0.000000 e r0.000000 f
r0.000000 g r0.000000 h

#10 r3.000000 a r2.916666 b r3.000000 c r0.000000 d r1.000000 e r0.000000 f
r1.000000 g r0.000000 h


#20 r3.000000 a
```

Figure 10 -. Example of power traces file generated by VIPPE.

The user needs to consider that enabling power tracing means simulation time increment.

7.6 Activity traces

VIPPE can provide a trace of activity parameters. The trace reports the value of the following activity parameters along time:

- # instructions
- # cpu usage
- …

These reports can be used by external tools and post-processing stages, for power tracing or other extra-functional property assessments[9].

For enabling the export of the traces, the user shall add and additional `--vippe-out activity_traces` switch as illustrated below:

- **vippe32.x --vippe-platform-xml xmlfiles/ --vippe-mapping-xml xmlfiles/ --vippe-so-path ./ --vippe-out activity_traces**

[9] Specifically, these activity traces have been employed for feeding power assessment via Zynq PS characterization functions provided by the OFFIS institute.

8 Co-Simulation with SystemC

VIPPE can be co-simulated with SystemC. This way, it is possible, for instance, to connect a VIPPE model with a model of a digital platform in SystemC, or a SystemC- AMS platform.

The basic scheme is summarized in Figure 11, which reflects the example provided in *$VIPPE_HOME/examples/systemc/vippe_sc_ex*. On the top-right hand side, a control application is represented. This application is modeled and run with VIPPE (in the example, mapped to the default mono-processor platform). The application communicates with a SystemC simulation via a simple API (the example uses two calls, *dig_read* and *dig_write*), which model the read/write to a memory mapped region.

In the left-hand side part of Figure 11, a module represents a "pure" SystemC model, used as test bench in this example. A SystemC process (position_proc) updates the "position_cv" SystemC signal (of double type) signal every fixed period. Another SystemC process (actuation_proc) reads periodically the actuation_cv signal (of double type). The "vsc_cosim_contrl" module structures the interface and mapping code which will require user customization for considering more input/outputs and customized memory maps[10].

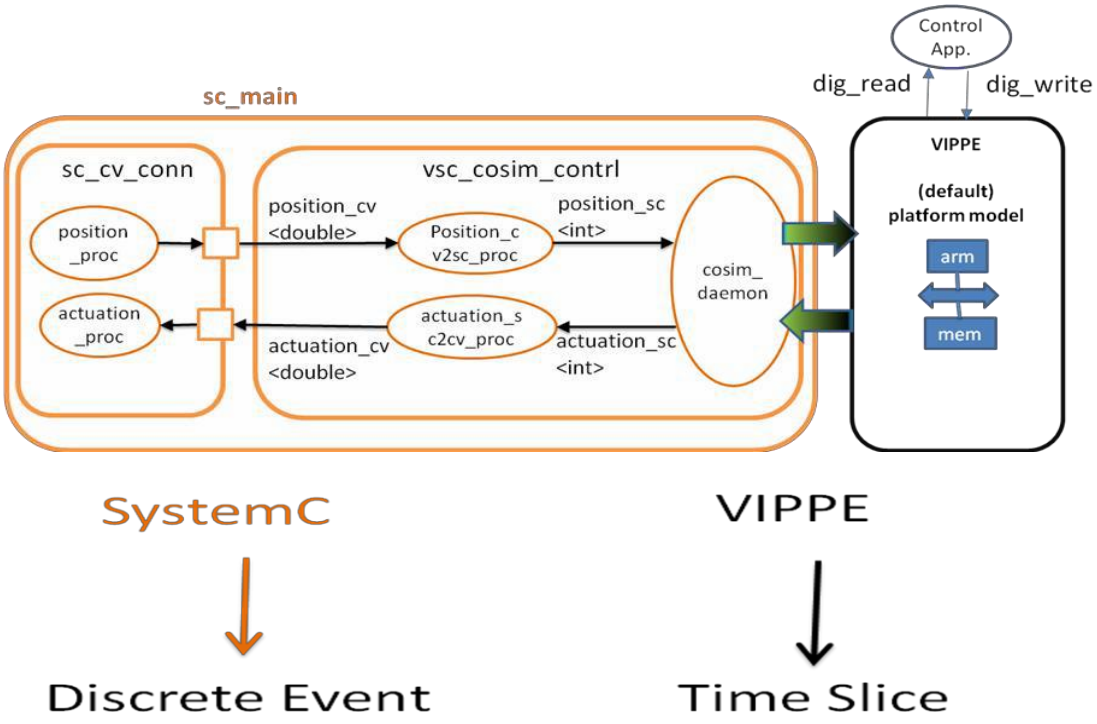


Figure 11 - Sketch of SystemC-VIPPE co-simulation in the *vippe_sc_ex* example available within the VIPPE distro.

[10] In next releases, VIPPE could include some generic template for this co-simulation interface. So far, the provided example can be used as a reference for user customized co-simulation models.

As the bottom part of Figure 11 reflects, the execution of this example is a co- simulation. In effect, there are two different simulation engines running: the SystemC simulation kernel and the VIPPE simulation kernel. SystemC simulation kernel runs a discrete-event (DE) simulation. VIPPE a fixed time-step simulation.

Following subsections explain how to compile and run the example, and in general a VIPPE-SystemC co-simulation, and the required insights for writing the application and the co-simulation interface.

8.1 Compiling and running the VIPPE-SystemC co-simulation

For compiling the example in *$VIPPE_HOME/examples/systemc/vippe_sc_ex* the user only needs to execute:

- **make**
- **make run**

By inspecting the Makefile and the example structure, it can be realized that, in general, the user describes and compiles independently the SystemC model and the VIPPE model.

At the root of the example, the *app.cpp* file where the application is described is required. Additionally, a XML files folder with the platform and mapping to platform description can be added and used, as was shown in section 8. For a clean separation, the SystemC model is described in a subfolder, that has to be called "*codsystemC*".

```
$VIPPE_HOME/examples/systemc/vippe_sc_ex$ ls -l
total 16
    app.cpp
    codsystemC
    Makefile
    platform_xml
```

The application is compiled has been shown in previous sections, e.g. producing a single executable file (called "*run.x*" in the example). The SystemC code will be compiled as any conventional SystemC code. The only requirement is to produce a SystemC executable called "*sc_run.x*".

```
$VIPPE_HOME /examples/systemc/vippe_sc_ex$ ls -l
    codsystemC/*.x
    codsystemC/sc_run.x
```

Once this file has been produced, the VIPPE simulation is launched with the `–vippe- systemc` switch.

```
$VIPPE_HOME /examples/systemc/vippe_sc_ex$ ./run.x –vippe-systemc
```

A run trace for the default configuration of the example should look like as shown in Figure 12.

```
./run.x --vippe-systemc
# Warning: No XML platform definition provided, using a default platform
          _____________________
         |       CPU0 (arm9)   |
         |         f =1GHz     |
         |         f =1GHz     |
         |_____________________|
             _____|_____
            |  IC | DC  |
            |  16 k |16 k |
            |_____|_____|
  /_____________|_____________\
 /          BUS1 1000000bps    \
 \_____________________________/
  \              |              /
           _____|_____
          |  MEMORY   |
          | taccess = 10ms |
          |    10ns   |
          |___________|
```

```
*** Start SYSTEMC simulation ***
Creating vsc_cosim_control instance vsc_cosim_ctrl
vsc_cosim_ctrl: READ_DATA, addr=0x8100, at (VIPPE) t=3110543ns, SystemC t=0 s actuation =0 at0 s
vsc_cosim_ctrl: WRITE_DATA, data=-75, addr=0x8200, at (VIPPE) t=24290805ns, SystemC t=20 ms vsc_cosim_ctrl: READ_DATA, addr=0x8100, at (VIPPE) t=24290814ns, SystemC t=24290805 ns vsc_cosim_ctrl: WRITE_DATA, data=75, addr=0x8200, at (VIPPE) t=34841884ns, SystemC t=30 ms vsc_cosim_ctrl: READ_DATA, addr=0x8100, at (VIPPE) t=34841893ns, SystemC t=34841884 ns vsc_cosim_ctrl: WRITE_DATA, data=75, addr=0x8200, at (VIPPE) t=45346399ns, SystemC t=40 ms vsc_cosim_ctrl: READ_DATA, addr=0x8100, at (VIPPE) t=45346408ns, SystemC t=45346399 ns vsc_cosim_ctrl: WRITE_DATA, data=75, addr=0x8200, at (VIPPE) t=55850914ns, SystemC t=50 ms vsc_cosim_ctrl: READ_DATA, addr=0x8100, at (VIPPE) t=55850923ns, SystemC t=55850914 ns vsc_cosim_ctrl: WRITE_DATA, data=75, addr=0x8200, at (VIPPE) t=66355429ns, SystemC t=60 ms vsc_cosim_ctrl: READ_DATA, addr=0x8100, at (VIPPE) t=66355438ns, SystemC t=66355429 ns vsc_cosim_ctrl: WRITE_DATA, data=75, addr=0x8200, at (VIPPE) t=76859944ns, SystemC t=70 ms vsc_cosim_ctrl: READ_DATA, addr=0x8100, at (VIPPE) t=76859953ns, SystemC t=76859944 ns vsc_cosim_ctrl: WRITE_DATA, data=75, addr=0x8200, at (VIPPE) t=87364459ns, SystemC t=80 ms vsc_cosim_ctrl: READ_DATA, addr=0x8100, at (VIPPE) t=87364468ns, SystemC t=87364459 ns
```

Figure 12 - Co-simulation trace of the vippe_sc_ex VIPPE-SystemC co-simulation example.

8.2 Writing the application for VIPPE-SystemC co-simulation

Table1 details the simple API which enables the direct access of software code to the SystemC part.

VIPPE API	Description
`unsigned dig_read (void* addr)`	Models the read of data from a specific memory address
`Void dig_write (void* addr, unsigned data)`	Models the write of data from a specific memory address
`void vippe_IRQ (void *)`	Registers a IRQ handler function

Table4. VIPPE API for letting an application code directly accessing SystemC.

By using this simple API, the platform dependent code accessing the device has to be slightly modified to model the driver[11]. Following, the application code of the example is reproduced. The code shows the modeling of a controller software working by polling. That is, there code runs in an endless loop (without any blocking synchronization). At each iteration, the code reads, though the `dig_read` function, from a fixed position (0x8100) a value from the hardware model (in SystemC), make some computation and then writes, through the `dig_write` function, the annotation value to another memory position (0x8200), where the actuator is mapped. The interface functions work with integer values. Any conversion required for SystemC model can be done in the interface code or in the SystemC model itself.

[11] A next step in VIPPE development will be to enable the modelling of unmodified driver code, by detecting accesses to memory mapped addresses and redirecting them to the SystemC model, e.g. to a TLM bus decoder.

```
#define N_ELEM 1000
#define ACTUATION_THRESHOLD 0
int main(int argc, char *argv[]) {
  printf("APP: (%lld ns) -- Start Function \n",time_real_watch());
  void* position_address=(void*)0x8100;
  void* actuation_address=(void*)0x8200;
  int position; // storing a fixed point value
  int actuation; // storing a fixed point value
  int a[N_ELEM], i, j, aux;
  printf("APP: (%lld ns) -- Start initialization \n",time_real_watch());
  for (i=0 ; i<N_ELEM ; i++)
       a[i] = N_ELEM-i;
       aux = 0;
       while(true) {
           printf("APP: (%lld ns) -- Start Read Position\n",time_real_watch());
           position = dig_read(position_address);
           printf("APP:(%lld  ns) -  End Read Pos.: %d \n", time_real_watch(), position);
           // model computational for actuation computation
           for (i=1 ; i<N_ELEM ; i++) {
               for (j=0 ; j<N_ELEM-i ; j++) {
                   if (a[j] > a[j+1]) {
                     aux = a[j];
                     a[j] = a[j+1];
                     a[j+1] = aux;
               }
           }
       }
       if(position>ACTUATION_THRESHOLD) {
           printf("APP:  (%lld  ns)  --  Write  actuation  :  value=    0.75\n",
time_real_watch());
           dig_write(actuation_address,75); // 0.75, with conversion  factor 100
       } else {
           printf("APP: (%lld ns) - Write actuation : value= -0.75\n", time_real_watch());
           dig_write(actuation_address,-75);  //  -0.75, with conversion factor 100
       }
  }
     printf("End of User code \n"); return 0;
}
```

Figure 13 - Excerpt of the code of the control application in the vippe_sc_ex example.

The computational load is calculated and simulated by VIPPE, therefore the time elapsed between the read and the subsequent write depends on several aspects, such as the amount of computational code, the underlying architecture (processor type, processor frequency, bus load, etc).

You can play with the code (e.g., change the number of iterations of the computation loop) or with the platform configuration (use your own platform instead the default one), to see the effect.

To see that effect, the code excerpt also shows the use of the `time_real_watch` function. This is a VIPPE API function which can be used for debugging and analysis purposes, which returns the current absolute VIPPE simulation time in nanoseconds. The user code can use timing services under any of the supported APIs. For instance, a POSIX time service like `gettimeofday`, so the user, unmodified code can be modeled and assessed in VIPPE.

8.3 VIPPE-SystemC Co-simulation interface

The "vsc_cosim_ctrl" module encloses the interface functionality. The provided example serves as a reference for a user customization. It shows the constructor of the co-simulation control module. The constructor retrieves a pointer to the shared memory IPC used for the co-simulation (VIPPE and SystemC simulation processes communicate via shared memory IPC). It also creates the SystemC process in charge of synchronizing with VIPPE ("cosim_daemon"), and two other SystemC processes which are used for a double-to-int and vice versa conversion for the position and actuation signals. The double-to-int conversion ("*position_cv2sc_proc*" process) multiplies the double value per a conversion integer factor and converts the result into an integer. This way, the integer can be used as a fixed-point value at the software side.

The inverse conversion (division by the conversion factor and conversion to double) is performed in the "*actuation_sc2cv_proc*" process. Notice that these conversion processes are activated only when the position signal (updated by the external environment modeled in SystemC) and the actuation signal (updated by the SW side) change.

```
vsc_cosim_control::vsc_cosim_control (char *shmem_id_par, sc_module_name name) :
sc_module(name) {
    // Creates shared memory IPC instance which serves for communication between VIPPE
    // and this SystemC co-simulation control block
     // defaults: set_real_fixpoint_conv_factor(100);
     shmem_id = shmem_id_par;
       // get pointer to the shared memory IPC used for the cosimulation
    systemc_conn = (systemc_conn_struct *) create_sh_mem (sizeof(systemc_conn_struct),
      INT_TO_SHARED_MEM_CREATE_SYSTEMC, shmem_id);
    SC_THREAD(cosim_daemon);
    SC_THREAD(position_cv2sc_proc); sensitive <<
    position_cv;   SC_THREAD(actuation_sc2cv_proc);
    sensitive << actuation_sc;
}
```

Figure 14 - Constructor of the co-simulation controller code in the vippe_sc_ex example.

However, the co-simulation control process is an endless loop, in charge of synchronizing the SystemC simulation with VIPPE simulation. This loop has a switch clause which contemplates the different events that required synchronization. The following code excerpt presents the two case statements that the user might need to customize for the simple synchronization scheme shown in Figure 11.

The co-simulation controller implements a conventional *locked* synchronization scheme. The synchronization is done in the following way. VIPPE time monotonically advances at a fixed time step called *time slice* (10ms in the default configuration). The co-simulation control process is in charge of ensuring that the SystemC simulation catches up this time, blocking VIPPE synchronization if required. That is why the co- simulation trace shown in Figure 12 always shows higher or equal VIPPE times.

The co-simulation controller code is loop which contains a *switch statement*, which contemplates all the possible types of VIPPE events that have to be notified to SystemC for synchronization purposes.

Figure 15 shows the three types of events that are considered in the *vippe_sc_ex* example.

The VIPPE time slice is used as synchronization wall. That is, in any case, the end of the VIPPE time slice is notified to SystemC. Then if there are not events of any other type from VIPPE to SystemC, this case will be taken, and the co-simulation controller forces a synchronization between VIPPE and SystemC (SystemC will catch up until that time). The `systemc_conn->next_time` shared variable is used to transfer the current VIPPE time to the SystemC side. It is converted to a SystemC time type. The case0 code assumes, as was mentioned, that SystemC time is behind the VIPPE time, so the next step is to calculate the time that the SystemC simulation needs to advance with respect to the current SystemC time (retrieved with `sc_time_stamp()`).

```
          case 0:
#ifdef _VSC_COSIM_CONTROL_VERBOSE
//                cout << name() << ": END OF SLICE SYNCHRONIZATION,
(advance SystemC) until (VIPPE) t= " << systemc_conn->next_time << "ns,
SystemC t=" << sc_time_stamp() << endl;
#endif
               nexttime=sc_time(systemc_conn->next_time, SC_NS);
               wait(nexttime-sc_time_stamp());
               systemc_conn->type=5;
               sem_post(&systemc_conn->semaphorescv);
               break;
         case 3:
#ifdef _VSC_COSIM_CONTROL_VERBOSE
               cout << name() << ": READ_DATA, addr= " << systemc_conn->addr
<< ", at (VIPPE) t= " << systemc_conn->next_time << "ns, SystemC t=" <<
sc_time_stamp() << endl;
#endif
               nexttime=sc_time(systemc_conn->next_time, SC_NS);
               wait(nexttime-sc_time_stamp());
```

```
            // here address decoding is performed
            if(systemc_conn->addr=(void *)0x8100) {
                systemc_conn->data = position_sc.read(); // Read  signal
mapped there
                systemc_conn->type=5;
            } else {
                SC_REPORT_ERROR("vsc_cosim_control","Bus  Error.  Reading
non-declared address");
            }
            sem_post(&systemc_conn->semaphorescv);
            break;
         case 4:
#ifdef _VSC_COSIM_CONTROL_VERBOSE
            cout << name() << ": WRITE_DATA, data=" << systemc_conn-
>data<< ", addr= " << systemc_conn->addr << ", at (VIPPE) t= " <<
systemc_conn->next_time << "ns, SystemC t=" << sc_time_stamp() << endl;
#endif
            nexttime=sc_time(systemc_conn->next_time, SC_NS);
            wait(nexttime-sc_time_stamp());

              // here address decoding is performed
              if(systemc_conn->addr=(void *)0x8200) {
                  // Here, data is available, in systemc_conn->data,
                  // and signal can be updated
                  actuation_sc.write(systemc_conn->data);
                  systemc_conn->type=5;
              } else {
                  SC_REPORT_ERROR("vsc_cosim_control","Bus  Error.  Writting
  at non-declared ddress");
              }

              sem_post(&systemc_conn->semaphorescv);
              break;
          default:
              SC_REPORT_ERROR("vsc_cosim_control","Type  of  cosim  message
  not supported");
      }
  }
}
```

Figure 15 - Excerpt of the co-simulation controller code in the vippe_sc_ex example with events from VIPPE to SystemC.

Case 3 reflects the case when VIPPE perform a read access. Case 4 reflects the case when VIPPE perform a write access. These events can happen, in general, in the middle of a VIPPE time slice, and involve a further synchronization between VIPPE and SystemC. The code excerpt shows address decoding in both cases. For it, the `systemc_conn->addr` shared variable is used. This variable is always settled by VIPPE (either by `dig_read` or by `dig_write`). Therefore, the co-simulation control process can read this data and perform any action in SystemC after address decoding.

The `systemc_conn->data` shared variable is used for the data transfer. In the case of the `dig_read` function, the co-simulation controller process can state the desired value (before the `sem_post(&systemc_conn->semaphorescv);` synchronization call), which will be returned by the `dig_read` call at the VIPPE application side. In the example, the value is loaded with the value of the SystemC "*position_sc*" signal. In the case4 (reached after calling the `dig_write` function), the `systemc_conn->data` shared variable will make available the data passed to the `dig_write` function.

As for the case0, the write and read accesses end the interface coding stating the `systemc_conn->type` shared variable to the value5, reserved for notifying an ACK response to the VIPPE side. This value5 has to be read by VIPPE side, and therefore is not expected (and has to be considered as an error) by the co-simulation control process.

As can be noticed, cases1 and2 are not used in this example. These codes are reserved for supporting read/write access notifications to a TLM bus.

8.4 Modelling interrupts sourced by SystemC

The VIPPE-SystemC co-simulation interface also supports the modeling of interrupts. This a specific case where the SystemC part (in charge of modeling the hardware part) produces events that have to be realized by the application.

For that, VIPPE enables the capture at the application level of the Interrupt Service Routine (ISR) and its registration. For it, VIPPE API provides the `vippe_IRQ` function, reported in Table4. This function should be invoked at the beginning of the main functionality. It takes as argument a function pointer to the ISR. Following, a simple example of declaring an ISR and registering it is shown. An example is found in the *$VIPPE/examples/systemc/dummytest* folder.

```
void my_ISR(){
  printf("User written IRS routine\n");

  return;
}


int main(int argc, char *argv[]) {
...
  // register the IRS
    vippe_IRQ(i, my_ISR);
...
}
```

9 Exploration Interface

VIPPE supports the interfacing with an exploration to for configuring an automated design space exploration loop. The exploration tool will decide the next configuration to simulate via stating the value of some DSE parameters. The exploration tool receives the output metrics values, which determine the "merit" of the analyzed or simulated configuration.

VIPPE supports the creation of a configurable and executable performance model. The model is configurable in the sense that it allows to fix the value of a number of predefined *DSE parameters* (the ones passed by the exploration tool), before launching the simulation.

In this scheme, VIPPE makes the role of the analysis tools, i.e. a performance analysis by means of simulation.

VIPPE supports the communication with the exploration tool by supporting the Multicube XML interface [14] .

The exploration tool can launch a vippe executable model regardless if it is a single executable file or it is run from the vippe launcher through a command with the same syntax:

- **VIPPE_EXE_NAME | vippe.x | vippe32.x --xml_system_configuration SC_FILE --xml_system_metrics SM_FILE --xml_metric_definition SMD_FILE**

Where:

VIPPE_EXE_NAME: refers to the name of the executable in the case a single executable file has been compiled.

vippe.x or vippe32.x launcher applications are used depending on the target wordwidth at the compilation time.

SC_FILE: is the name of the Multicube XML file containing the configuration to be simulated

SM_FILE: is the name of the Multicube XML file containing the output metrics corresponding to the simulated configuration.

SMD_FILE: is the XML file containing the output metrics that are asked to be reported.

10 Measuring Function Response times

An interesting utility of VIPPE can be for characterizing the response time of a function or functions, e.g. to extract a distribution function of the response times of a function for a given target processor. Following subsections distinguish two different techniques, suitable for difference scenarios.

10.1 Instrumenting a function

A simple approach consists relies on the capture of the function in a separate source file, in order to instrument only the compilation unit associated to that file. The rest of the program is compiled with no instrumentation.

An example is provided in the *$VIPPE_HOME/examples/accuracy/multifile* example, specifically through the "mix" and "run_mix" rules. The "file2.cpp" source file is compiled with the ASM instrumentation method, while the invoking code "main.cpp" is compiled without instrumentation. Therefore, the code of main.cpp can contain a model of stimuli, e.g. random stimuli, and VIPPE can be used to estimate the response time after each invocation.

The response times estimated are representative as long as the function is mapped to a processor without caches and with no other interference sources (otherwise VIPPE will consider them).

10.2 In a fully instrumented scenario

The aforementioned method is not that interesting if the user desires to estimate the response times of a function after each invocation in the working environment of the function. That is, in an environment where the function is stimulated by other functions of the container program, which in turn is stimulated by an environment model (I/O in the actual world). In such a realistic environment, a function can present different response times for invocations with the very same input data, simply because the state of the caches and of the bus are different at the beginning of each invocation.

For such a case, VIPPE provides an additional feature, consisting in the possibility to:

- Add additional instrumentation to measure the response times of the functions at compilation time
- Report those report times after simulation.

For the former, the VIPPE compiler has three possible options:

--vippe-ftimes: adds instrumentation at the beginning and end of functions for response time measurement

--vippe-fcaches: adds instrumentation for accounting cache statistics within function context

--vippe-fannotate: covers the previous two options

The user shall compile the code with one of the aforementioned flags. Then, at execution time, the following switch can be used:

--vippe-out [dependencies | callgrind | function_times_list | function_caches_list | response_times]

For instance, if the user compiles VIPPE executable model called "run.x", then the following command:

```
$run.x –vippe-out response_times
```

Creates a file called *"response_times.out.vippe".* This file contains a list of functions (one per raw). First column reports the function name. The following columns report the minimum, maximum and average response times. The last column reports the number of calls which happened in the simulation thread.

The following command:

```
$run.x –vippe-out function_times_list
```

shows a more detailed report. This command generates a file called *"functEstim.out.vippe".* This file reports in plain text a sequence of function calls, with their begin, end and elapsed times for the invocation, as they are invoked by the VIPPE user process.

A simple example showing how to compile and execute for getting these reports is available in *$VIPPE_HOME/examples/function_analysis/bubble_ftimes*.

The example in *$VIPPE_HOME/examples/function_analysis/bubble_so_ftimes* shows the same when the application is run from the vippe launcher application.

11 Usage Examples

11.1 Samples List

The VIPPE distribution integrates some simple examples for a first introduction into the library. The following table provides a summary for the examples included in *$VIPPE_HOME/examples* dir, referred in this user guide and which functionalities can be explored at least with that example.

No.	Example	Path relative to $(VIPPE_HOME)	Aspects
1	bubble	Accuracy / bubble	Basic usage. VIPPE compiler. Flags for compilation with/without caches, for verbose mode.
2	bubble_sc_com p	Accuracy / bubble_sc_com	Compilation with the sc compiler
3	bubble_gui	accuracy/bubble_gui	GUI
4	bubble_so	accuracy/bubble_so	Instrumented code as .so library. VIPPE launcher
5	multifile	accuracy / multifile	Separated compilation. VIPPE3 types of compilation methods. Merging of instrumentation methods.
6	bubble_ftimes	function_analysis / bubble_ftimes	Functional call and response time reports
7	bubble_so_ftimes	function_analysis / bubble_so_ftimes	Functional call and response time reports with vippe launcher and so's.
8	vippe_sc_ex	systemc/vippe_sc_ex	SystemC interface

11.2 Tutorials

Let's run some included examples of good work with this VIPPE distribution.

11.2.1 Bubble

First, in the linux terminal we set into <*(VIPPE_HOME)/examples/accuracy/bubble*> dir and type:

- vippe-llvm++ bubble.cpp -o bubble.o

We compile our source file <*bubble.cpp*> into the <*bubble.o*> object file with Vippe compiler. Also, vippe has created an instrumentation file called <*vippe_bubble_opt_del.bc*> with the annotations for the simulator required metrics. Now, we use clang to apply the rest of the optimization passes:

- clang++ -g -O0 bubble.o -o run.x -I$VIPPE_HOME/include -L$VIPPE_HOME/lib -lvippeposix -lvippe -lpthread -lrt -lxml2

And finally, we can run our executable just typing <*./run.x*>. VIPPE simulates the program, showing used architecture and results information.

11.2.2 Bubble ftimes

Now let's see the times function annotation functionality. In the linux terminal we set into <$*(VIPPE_HOME)/examples/function_analysis/bubble_ftimes*> directory and type:

- vippe-llvm++ --vippe-ftimes -O2 --vippe-asm --vippe-cpu=armv7a bubble.cpp -o bubble.o -m32

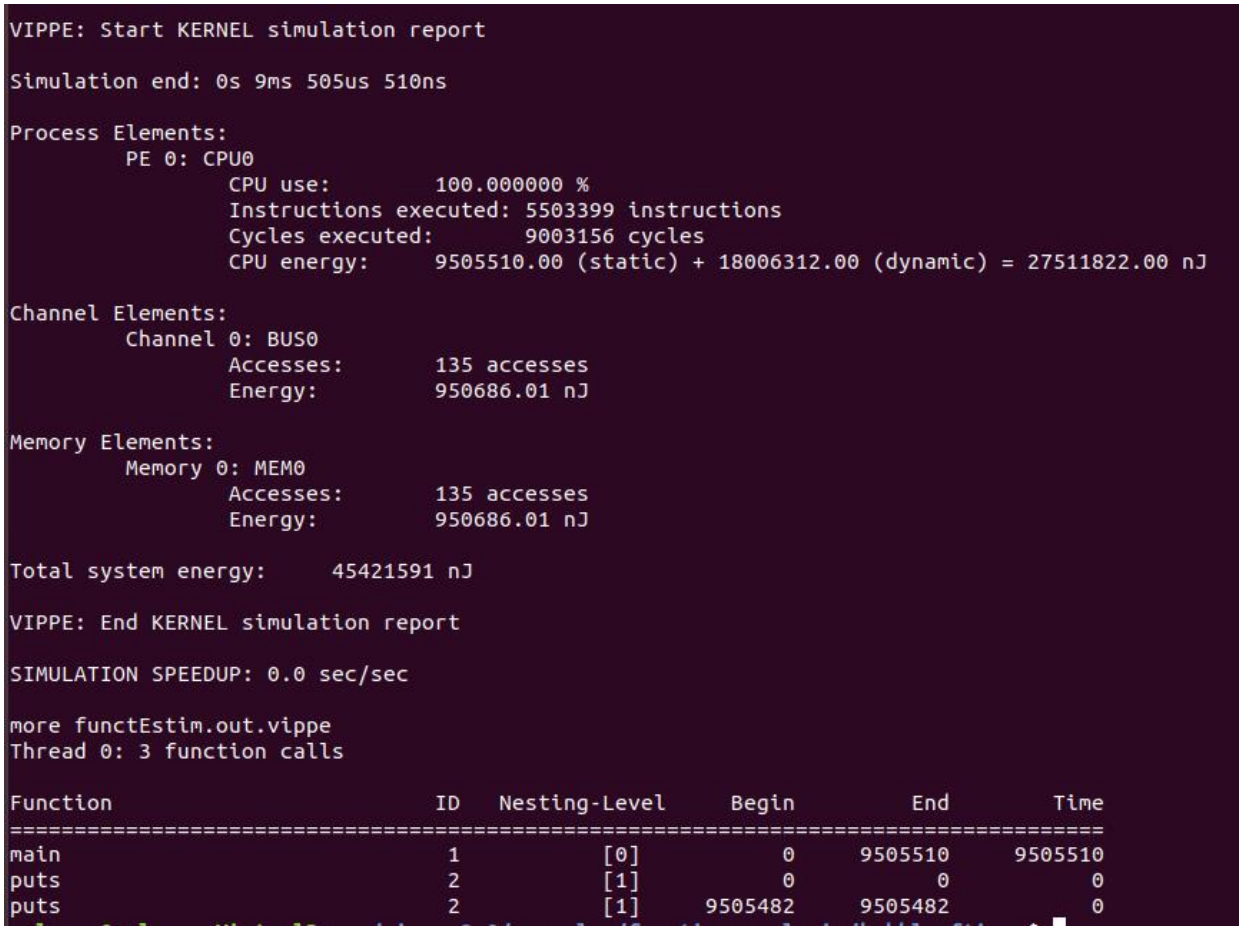
So, we compile our source file <*bubble.cpp*> into the <*bubble.o*> object file with Vippe compiler. Also, vippe has created an instrumentation file called <*vippe_bubble_opt_del.bc*> with the annotations for the simulator required metrics, and <*bb_analysis_file.vippe*> with time function annotations. Now, we use clang to apply the rest of the optimization passes:

- clang++ -m32 -g -O0 bubble.o -o run.x -I$VIPPE_HOME/include -L$VIPPE_HOME/lib -lvippeposix32 -lvippe32 -lpthread -lrt -lxml2

Finally, we can run our executable just typing:

- ./run.x --vippe-out function_times_list more functEstim.out.vippe

As result VIPPE simulates the program and shows additional function annotation information.



```
VIPPE: Start KERNEL simulation report

Simulation end: 0s 9ms 505us 510ns

Process Elements:
         PE 0: CPU0
                 CPU use:            100.000000 %
                 Instructions executed: 5503399 instructions
                 Cycles executed:       9003156 cycles
                 CPU energy:     9505510.00 (static) + 18006312.00 (dynamic) = 27511822.00 nJ

Channel Elements:
         Channel 0: BUS0
                 Accesses:           135 accesses
                 Energy:             950686.01 nJ

Memory Elements:
         Memory 0: MEM0
                 Accesses:           135 accesses
                 Energy:             950686.01 nJ

Total system energy:     45421591 nJ

VIPPE: End KERNEL simulation report

SIMULATION SPEEDUP: 0.0 sec/sec

more functEstim.out.vippe
Thread 0: 3 function calls

Function                        ID   Nesting-Level     Begin         End         Time
=========================================================================================
main                             1          [0]           0     9505510     9505510
puts                             2          [1]           0           0           0
puts                             2          [1]     9505482     9505482           0
```

11.2.3 Vippe_sc_ex

Now let's see the SystemC interface functionality. In the linux terminal we set into <$*(VIPPE_HOME)/examples/*systemc/vippe_sc_ex> directory and type:

- vippe-llvm++ --vippe-verbose --vippe-preserve-files -I$VIPPE_HOME/include app.cpp -o app.o

So, we compile our source file <*bubble.cpp*> into the <*bubble.o*> object file with Vippe compiler. Also, vippe has created an instrumentation file called <*vippe_bubble_opt_del.bc*> with annotations for the simulator required metrics. Now use clang to apply the rest of the optimization passes:

Generating LLVM intermediate code, typing:

- clang++ -emit-llvm -c -I$VIPPE_HOME/include/ -include VIPPEdefines.h app.cpp -o vippe_app.bc -I $VIPPE_HOME/include

Applying LLVM optimizations:

- opt -load $VIPPE_HOME/utils/llvm_install/lib/LLVMVippe.so -cxxcounter < vippe_app.bc > vippe_app_opt.bc

Generating LLVM intermediate code of icache functions:

- clang++ -emit-llvm -c uc_vippe_icache_tmp_file.cpp -o uc_vippe_icache_tmp_file.bc -I$VIPPE_HOME/include/

Linking bc files:

- llvm-link $VIPPE_HOME/lib/inline_functions.bc vippe_app_opt.bc uc_vippe_icache_tmp_file.bc -o vippe_app_union.bc

Applying LLVM optimizations for global constructors:

- opt -load $VIPPE_HOME/utils/llvm_install/lib/LLVMVippe.so -cxxdelextlink
  < vippe_app_union.bc > vippe_app_opt_del.bc

Applying std VIPPE optimizations:

- opt -std-compile-opts vippe_app_opt_del.bc -o vippe_app_union_opt.bc

Compiling file:

- clang++ -c vippe_app_union_opt.bc -o app.o -fPIC
- clang++ -g -O0 app.o -o run.x -I$VIPPE_HOME/include -L$VIPPE_HOME/lib -lvippeposix -lvippe -lpthread -lrt -lxml2

Finally, we can run our executable just typing

- ./run.x --vippe-systemc

As result VIPPE simulates the program and shows additional SystemC interface information.

```
# Warning: No XML platform definition provided, using a default platform
      _____________
     | CPU0 (arm9) |
     |   f = 1GHz  |
     |_____________|
     |  IC  |  DC  |
     | 16 k | 16 k |
     |_____________|
            |
 /__________|_________________\
/       BUS  11000000bps      \
\_____________________________/
 \        |                  /
    ______|______
   | MEMORY      |
   |  taccess =  |
   |        10ns |
   |_____________|

codsystemC/sc_run.x 903573
APP: (10 ns) -- Start Function
APP: (20 ns) -- Start initialization

            SystemC 2.3.0-ASI --- Mar 11 2020 17:18:17
        Copyright (c) 1996-2012 by all Contributors,
        ALL RIGHTS RESERVED

*** Start SYSTEMC simulation ***
Creating vsc_cosim_control instance vsc_cosim_ctrl
actuation = 0 at 0 s
APP: (16360 ns) -- Start Read Position
vsc_cosim_ctrl: READ_DATA, addr= 0x8100, at (VIPPE) t= 16360ns, SystemC t=0 s
APP: (16360 ns) -- End   Read Position : value = 0
APP: (43028352 ns) -- Write actuation : value= -0.75
vsc_cosim_ctrl: WRITE_DATA, data=-75, addr= 0x8200, at (VIPPE) t= 43028352ns, SystemC t=40 ms
APP: (43028378 ns) -- Start Read Position
vsc_cosim_ctrl: READ_DATA, addr= 0x8100, at (VIPPE) t= 43028378ns, SystemC t=43028352 ns
APP: (43028378 ns) -- End   Read Position : value = 5
APP: (64062370 ns) -- Write actuation : value=  0.75
vsc_cosim_ctrl: WRITE_DATA, data=75, addr= 0x8200, at (VIPPE) t= 64062370ns, SystemC t=60 ms
APP: (64062396 ns) -- Start Read Position
vsc_cosim_ctrl: READ_DATA, addr= 0x8100, at (VIPPE) t= 64062396ns, SystemC t=64062370 ns
APP: (64062396 ns) -- End   Read Position : value = 7
APP: (85096388 ns) -- Write actuation : value=  0.75
vsc_cosim_ctrl: WRITE_DATA, data=75, addr= 0x8200, at (VIPPE) t= 85096388ns, SystemC t=80 ms
APP: (85096414 ns) -- Start Read Position
vsc_cosim_ctrl: READ_DATA, addr= 0x8100, at (VIPPE) t= 85096414ns, SystemC t=85096388 ns
APP: (85096414 ns) -- End   Read Position : value = 10
APP: (106130406 ns) -- Write actuation : value=  0.75
vsc_cosim_ctrl: WRITE_DATA, data=75, addr= 0x8200, at (VIPPE) t= 106130406ns, SystemC t=100 ms
APP: (106130432 ns) -- Start Read Position
```

12 References

[1] L. Diaz, P. Sánchez. "Host-compiled Parallel Simulation of Many-core Embedded Systems". San Francisco, DAC2014. 2014-06.

[2] L. Diaz, E. González, E. Villar, P. Sánchez. "VIPPE: Parallel simulation and performance analysis of complex embedded systems". HiPPES4CogApp: High Performance, Predictable Embedded Systems for Cognitive Application. 2015-01

[3] L. Diaz, E. González, E. Villar, P. Sánchez. "VIPPE: Native simulation and performance analysis framework for multi-processing embedded systems". Proceedings of the JCE-Sarteco2014. 2014-09.

[4] CONTREX FP7 project site: https://contrex.offis.de/home/

[5] QUEMU website. http://wiki.qemu.org/Main_Page. Last visit. Oct., 2014.

[6] OVP website. http://www.ovpworld.org/. Last visit. Oct., 2014.

[7] L. Diaz, P. Sánchez. "Host-compiled Parallel Simulation of Many-core Embedded Systems". San Francisco, DAC2014. 2014-06

[8] J. Castillo, H. Posadas, E. Villar, M. Martínez. “Energy Consumption Estimation Technique in Embedded Processors with Stable Power Consumption based on Source-Code Operator Energy Figures”. In proceedings of XXII Conference on Design of Circuits and Integrated Systems, DCIS'07. Nov., 2007

[9] S. Real, H. Posadas, E. Villar. “L2 Cache Modeling for Native Co- Simulation in SystemC”. Symposium of Industrial Embedded Systems (SIES2010). Jun. 2010

[10] Coremu website. http://sourceforge.net/p/coremu/home/Home/. Last visited, August, 2015.

[11] A.Wang et al. “COREMU: A Scalable and Portable Parallel Full-system Emulator”. In PPoPP’11, February12–16,2011, San Antonio, Texas, USA.

[12] http://www.greensocs.com/get-started#qbox

[13] Imperas SW Limited. Imperas Installation and Getting Started Guide. Version1.4.25.3. Oct.2014.

[14] Vittorio Zaccaria and Gianluca Palermo . “Specification of the XML interface between design tools and use cases. R1.4”. Oct.1st, 2009. Available in http://www.multicube.eu/public_docs.html.

[15] GESE/UC UML/MARTE modeling methodology. Available in http://umlmarte.teisa.unican.es (see Documentation section). Last visited June, 22th, 2016.

[16] CONTREX Eclipse Plug-in site. http://contrep.teisa.unican.es . Last visited June, 22th, 2016.