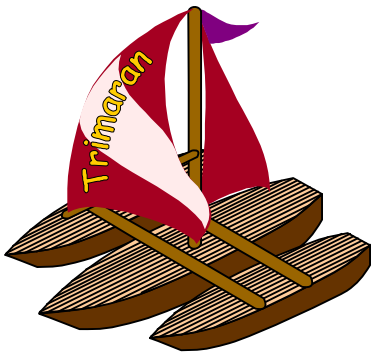

The Simulation and Performance Monitoring Environment of Trimaran

This chapter describes the structure and operation of the Trimaran HPL-PD simulation environment. It also describes the performance monitoring framework interface, and the development of analysis tools.

Table of Contents

1. **Introduction**
2. **Components of the Simulator**
3. **What has changed**
4. **Installation requirements**
5. **Environment Variables required for installation and usage**
6. **Building the simu directory**
7. **Configuring the simulator**
8. **Using the simulator**
9. **Limitations**
10. **Building tools from the Performance monitoring framework**
11. **Configuring the performance monitor**
12. **Building performance monitoring tools**
13. **Appendix**
14. **References**

1. Introduction

The goal of the HPL-PD simulation environment is to

- Convert REBEL to executable code and emulate the execution on a virtual HPL-PD processor.
- Generate run-time information such as clock cycles taken for execution, average number of operations executed per cycle, etc.
- A framework to build tools that allow the gathering of run-time information, profiling, etc.

Design goals are

- ***VLIW emulation of HPL-PD:*** The scheduling and latency information appears in the execution stream of instructions itself. The instruction stream is interpreted by a HPL-PD virtual machine which is present in form of a library(libequals.a) that needs to be linked to. The emulator follows both the EQUALS and LTE scheduling model. For more information on these scheduling models, see [2].

- ***Platform independent design:*** The simulation makes little assumptions about the host platform operating system and the machine architecture. Platform independence comes mostly due to the fact that the HPL-PD emulatable is a low-level C file. This file is compiled and linked with the HPL-PD emulator code to form the final executable binary.

- ***Interoperability:*** To allow free inter-mixing of HPL-PD and host platform code. In other words, an executable program can consist of a mix of HPL-PD emulated and host platform object(.o) files. Thus, operating system / library functions can be called from the emulation environment.

- ***Hooks for performance monitoring:*** Allow a developer to build tools that perform run-time execution analysis and profiling. Other examples of such tools include a memory cache emulator, and memory dependences analysis. This is possible by emitting run-time information as an execution trace and then building a framework that allows building analysis tool by filtering out irrelevant information from the trace.

2. Components of the Simulator

- ***Code generator (trimaran/simu/bin/codegen)***

This module generates a low-level C file similar to an assembly file from Elcor's intermediate representation. The file is then compiled instead of being assembled. C is the language of choice for this assembly-equivalent file to provide complete platform independence. In other words, this file can be compiled on any platform without any modifications.

Four files are created:

- ***A file with the .inc extension*****:** contains external variable declarations, global data, modified structure and union layout from the original C source - this information is required for inter-operation with native code, other global data to hold information required by the emulator at execution time.

- ***A file with the .tbl extension:*** a collection of emulation tables. An emulation table holds all the HPL-PD machine operations required by the application. One emulation table is maintained per C procedure of the original C source.

- ***A file with the .c extension:*** includes the .inc and .tbl files. It also contains a series of functions which serve as stub routines to inter-operate with native / host-platform code.

- ***A file named benchmark_data_init.simu.c:*** calls function in the compiled files that initialize the program's global data

- ***Emulation Library (trimaran/simu/lib/libequals.a)***

The library consists of the HPL-PD virtual machine - an interpreter and a set of emulation routines form the HPL-PD virtual machine. The interpreter is invoked on every procedure entry. It emulates the instruction stream in a loop until the procedure returns.

There is one emulation function for every HPL-PD operation. These emulation functions are automatically generated from the operation specification. The specification of HPL-PD operations consists of its I/O format and its actions.

The specification is present in **trimaran/simu/src/Emulib/ops.list**. The generated emulation routines and the interpreter code are also present in the same directory.

"**make genops**" generates the operations.
"**make lib**" builds the library.
"**make**" will generate the operations and then build the library

Note that the generated library by default includes speculation support.
Refer to **trimaran/simu/src/Emulib/Makefile** for details.

- ***Performance Monitoring Library (trimaran/simu/bin/libperf.a)***

This library provides a C++ interface for building of performance monitoring tools. Examples of such tools can be a data address cache emulator, a memory profiler, or a control flow profiler.

The performance monitoring framework (PM) processes events generated by the HPL-PD simulator and filters them out providing them in a rich C++ scan-able form. The PM framework has been explained in detail below.

3. What has changed

Codegen was entirely re-implemented. **Trimaran 1.0 Codegen** generated files are no longer supported and are not compatible with the new **Trimaran 2.0 emulation library**. Similarly, the emulation library has changed considerably (i.e. bug fixes, speculation support, ...) and is not compatible with any **Trimaran 1.0 Codegen** generated files.

Below is a very short and brief description of the implemented changes:

- The emulation tables are no longer initialized at run-time - global table initializations are now used
- New Makefiles, with dependency support
- Enhanced performance monitoring utility
- Resolved several bugs in function wrapper generation
- Implemented vararg and stdarg support
- Resolved dynamic statistic count error
- Resolved several bugs in Emulation environment initialization
- Resolved bug in SAVE/RESTORE operations
- Implemented Code Speculation support
- Scripts for Elcor to Simu to Binary conversion

4. Installation requirements

Installing Codegen and Perf (the Performance monitoring tools):

Codegen can be compiled using either "gcc" version 2.7.2.1 (or greater) or HP's ansi C++ compiler, "aCC" version A.01.12 (for HP-UX B.10.10 and B.10.20). It may work with other ANSI C++ compilers but has not been tested.

Codegen includes some **impact** and **elcor** libraries, and hence, **Codegen**, **elcor** and **impact** have to be compiled with compatible compilers. If you use "aCC" for **elcor**, then you must use "aCC" for **Codegen**. The environment variables, CXX for C++ compiler controls the compiler that is used to build **elcor**, **Codegen** and **Perf** - refer to the installation instructions in **trimaran/elcor** and **trimaran/impact** for more information on how the **Impact** and **Elcor** modules should be compiled.

In addition, building **Perf** requires gnu's "ar" and "ranlib" to build the **libperf.a** library and "gnumake" to run the **Perf** makefiles. Please make sure that they are available in your search path.

Installing Emulib:

The *Emu*lation *Lib*rary can be compiled with either gcc, or cc. In addition, building **Emulib** requires gnu's "ar" and "ranlib" to build the **libequals**.library and "gnumake" to run the **Emulib** makefile. Please make sure that they are available in your search path.

Shared and static versions of the library can be compiled. By default, a static library is created. If a shared library is desired, define **GEN_SHARED_LIBRARY** in your environment.

5. Environment Variables required for installation and usage

Before building the simulator directory, the following environment variables should be defined to their appropriate values as shown.

Compilation variables
setenv TRIMARAN_HOST_TYPE <type of machine you are compiling on, 'hp' or 'x86lin'>
setenv CC gcc
setenv CXX gcc

IMPACT variables
setenv IMPACT_REL_PATH <impact_distn_dir>
setenv STD_PARMS_FILE $IMPACT_REL_PATH/parms/STD_PARMS.TRIMARAN

ELCOR variables
setenv ELCOR_REL_PATH <elcor_distn_dir>
setenv ELCOR_PARMS_FILE $ELCOR_REL_PATH/parms/ELCOR_PARMS

REACT-ILP variables
setenv SIMU_REL_PATH <simu_distn_dir>
setenv SIMU_PARMS_FILE $SIMU_REL_PATH/parms/SIMULATOR_DEFAULTS

if a shared emulation library is desired
setenv GEN_SHARED_LIBRARY

6. Building the *simu* directory

The target "all" in the distribution directory **$SIMU_REL_PATH** builds the simulation environment as follows:

1. Creates the directories: **$SIMU_REL_PATH/lib, $SIMU_REL_PATH/src/Emulib/static_lib** and **$SIMU_REL_PATH/src/Emulib/shared_lib**
2. It builds the dependences in **$SIMU_REL_PATH/src/Codegen** directory
3. It builds **codegen** in *the **$SIMU_REL_PATH/src/Codegen*** directory. This builds and installs the binary in **$SIMU_REL_PATH/bin**
4. It builds the emulation library in **$SIMU_REL_PATH/src/Emulib**. The library is eventually installed in **$SIMU_REL_PATH/lib**
5. It builds dependences in the **$SIMU_REL_PATH/src/Perf/Tracer**, as well as in the example analysis applications directory **$SIMU_REL_PATH/src/Perf/Clients**
6. It builds the performance monitoring library and a couple of example programs, and installs them in **$SIMU_REL_PATH/lib** and **$SIMU_REL_PATH/bin** respectively

7. Configuring the Simulator

The simulator parameters can be set either through the Trimaran graphical user interface, or by directly modifying the file **trimaran/simu/parms/SIMULATOR_DEFAULTS.**

Note that the parameters described below can also be applied to **codegen** directly through the **-F** switch. After every change in the SIMULATOR_DEFAULTS file, simulation files need to be regenerated.

Following is a list of the simulator parameters that can be set, with a brief explanation of each parameter:

Parameter name: nual_equals
Value: yes | no

Determines which latency model will be used. If set to yes then the *equals* (EQUALS) model is used. If set to no, then the less-than-or-equals (LTE) model is used. The LTE model is simulated by setting all operation latencies to one clock cycle.

Parameter name : emulate_unscheduled
Value: yes | no

This parameter should be set to yes if either prepass or postpass scheduling are turned off in the Elcor compiler. This allows the emulation of unscheduled code.

Parameter name : emulate_virtual_regs
Value: yes | no

Setting this parameter to yes causes the simulator to emulate virtual registers, so that registers are created as needed. This feature is necessary when register allocation has been turned off in the Elcor compiler.

Parameter name : performance_monitoring
Value: yes | no

This parameter must be set to yes to allow trace information to be generated during simulation. The content and format of the trace output are controlled by the related parameters described below. The file output when performance monitoring is enabled is described in detail in section 8. The performance monitoring framework (PM) is described in detail in section 10.

Parameter name : control_flow_trace
Value: yes | no

Setting this parameter to yes causes control flow information to be included in the trace file (**DYN_TRACE**) generated during simulation. Control flow information includes procedure entry events, control block entry events, and operation nullification events. The contents of the **DYN_TRACE** file are described in detail in section 8. The performance_monitoring parameter must be set to yes for this parameter to have any effect.

Parameter name : address_trace
Value: yes | no

Setting this parameter to yes causes memory access (load / store) information to be included in the trace file (**DYN_TRACE**) generated during simulation. The contents of this file are described in detail in section 8. The performance_monitoring parameter must be set to yes for this parameter to have any effect.

Parameter name : binary_trace_format
Value: yes | no

Setting this parameter to yes emmitts trace information in binary form. Otherwise ASCII text is used. The ASCII format can be used in debugging. The performance_monitoring parameter must be set to yes for this parameter to have any effect.

Parameter name: dynamic_stats
Value: yes | no

When this parameter is set to yes, the simulation generates a dynamic statistics file (**DYN_STATS**) containing run-time execution information. The contents of this file are described in detail in section 8. The performance_monitoring parameter must be set to yes for this parameter to have any effect.

8. Using the Simulator

Refer to the Trimaran C Compiler for information on compiling a benchmark (**trimaran/bin/tcc**).

The simulator can produce two output files, the **DYN_STATS** file and the **DYN_TRACE** file. The simulator produces output when the parameter **performance_monitoring** has been enabled.

- ***The DYN_STATS file:***

If, in addition to the **performance_monitoring** parameter, the parameter **dynamic_stats** is enabled, the simulator produces a file called **DYN_STATS**. This file contains various run-time information collected from the execution of the program. This file is used by the Trimaran GUI to display the simulation results in various formats. Following is a sample **DYN_STATS** file.

 # Name of the procedure(name of the rebel file)
Function _main (eight_bb.el)

 # basic block, id, total scheduling length of the block
bb 1 dyn cyc: 1.00 sched len: 1
bb 2 dyn cyc: 2.00 sched len: 2
bb 3 dyn cyc: 1867.00 sched len: 10
bb 4 dyn cyc: 67.00 sched len: 1
bb 7 dyn cyc: 134.00 sched len: 1
bb 9 dyn cyc: 401.00 sched len: 3
bb 10 dyn cyc: 4.00 sched len: 5
bb 11 dyn cyc: 0.00 sched len: 3
bb 5 dyn cyc: 865.00 sched len: 7
bb 6 dyn cyc: 67.00 sched len: 2
bb 8 dyn cyc: 66.00 sched len: 2

 # Total number of HPL-PD cycles spent in executing this procedure
Dynamic_total_cycles: 3474.00

 # Same as above
Dynamic_total_compute_cycles: 3474.00

 # Not used for now
Dynamic_scalar_compute_cycles: 0.00 (0.00)
Dynamic_loop_compute_cycles: 0.00 (0.00)

 # Total number of operations executed in this procedure
Dynamic_total_operations: 3416.00

 # The break-up of operations and percentages in parentheses

 # total number of branch operations
Dynamic_branch: 666.00 (19.50)

 # total number of memory loads

Dynamic_load: 0.00 (0.00)

 # total number of memory stores
Dynamic_store: 2.00 (0.06)

 # total number of interger-alu operations
Dynamic_ialu: 1546.00 (45.26)

 # total number of floating point-alu operations
Dynamic_falu: 0.00 (0.00)

 # total number of compare operations
Dynamic_cmpp: 533.00 (15.60)

 # total number of prepare to branch operations
Dynamic_pbr: 668.00 (19.56)

 # Average number of operations executed per cycle
Dynamic_average_issued_ops/cycle: 0.98

 # Same as above, but the static breakup
Static_total_operations: 47.00
Static_branch: 6.00 (12.77)
Static_load: 0.00 (0.00)
Static_store: 2.00 (4.26)
Static_ialu: 26.00 (55.32)
Static_falu: 0.00 (0.00)
Static_cmpp: 3.00 (6.38)
Static_pbr: 8.00 (17.02)

 # Number of extra operations added by the register allocator
Dynamic_regalloc_op_overhead: 0.00 (0.00)

 # Same as register allocation overhead
Dynamic_spill_code: 0.00 (0.00)

 # Caller-saved register allocation overhead
Dynamic_caller_save: 0.00 (0.00)

 # Callee-saved register allocation overhead
Dynamic_callee_save: 0.00 (0.00)

 # Same as above, but the static breakup
Static_regalloc_op_overhead: 0.00 (0.00)
Static_spill_code: 0.00 (0.00)
Static_caller_save: 0.00 (0.00)
Static_callee_save: 0.00 (0.00)

- ***The DYN_TRACE file:***

If the parameter **control_flow_trace** or **address_trace** is enabled, an execution trace is created. The trace contains enough information to reproduce the complete execution state of an application. The trace is written out to the file **DYN_TRACE**.

Turning on control_flow_trace causes control-flow information to be included in the **DYN_TRACE** file. The events that are recorded in the trace are:

- ***Procedure entry:*** Entry into a procedure.
- ***Control-block entry:*** Entry to a basic block or a hyper block or a loopbody.
- ***Procedure exit:*** Exit from a procedure.
- ***Operation nullification:*** When an operation is nullified due to the effect of the guarded predicate operand in an operation.

Setting the **address_trace** to yes allows memory access events to appear on the trace. These constitute all the loads and stores in the program.

Following is an example of an excerpt from a trace in ASCII format. This trace includes both control flow and memory access information.

; procedure _mm_init entered
p _mm_init

; control block 14 entered, exit control flow edge was op 63 in the last
; block.
c 14 63

; control block 5 entered, fell thru in this block
c 5 0

; procedure return

; op 81 loaded a word from the address 40324cb8
LW 81 40324cb4

; op 83 was nullified by a guarded predicate
! 83

9. Limitations

The simulator has been extensively tested on HPUX and LINUX platforms, with a large suite of benchmarks such as SPECINT 92, SPECINT 95, and MEDIABENCH.

Note: Gcc 2.7.2 and 2.8.1 were used exclusively for benchmark certification.

- Currently, the simulator supports "varargs" and "stdargs" on LINUX platforms as well as HPUX platforms. The simulator makes some platform dependent assumptions, please report any bugs encountered.

- Benchmarks which include "setjmp/longjmp" are supported, although it is required that any function which includes a longjmp or a setjmp must either be strictly and exclusively compiled in the native "C" domain, or exclusively in the HPL-PD domain (i.e. jumps to HPL-PD code from native C code are not allowed).

- The emulation library includes full Data and Control Speculation. By default, libequals.a has minimal speculation support, i.e. exceptions are suppressed for all operations. For more information, refer to the main
- simulation loop in trimaran/simu/src/Emulib/PD_main.c.

- The performance monitoring utility currently does not support setjmp/longjmp events.

10. Building tools from the Performance Monitoring Framework

The simulator is the producer of information and the Performance monitoring tools are the consumers of this information. The communication channel between the consumers and the producers is the trace. Multiple trace consumers and the simulator can be run together as in shown in the example in the appendix.

The performance monitoring framework (or PM for short) provides an interface to build Performance monitoring tools. The basic infrastructure that processes and filters out information from the trace is present in **trimaran/simu/lib/libperf.a**.

The pre-requisites for using the Performance monitoring framework are:

- Performance monitoring option must be enabled.
- b) Control flow trace and/or address trace generation have to be enabled.

Note: The PM and the simulator share the same parameter files.

11. Configuring the performance monitor

The trace contains events generated from the entire simulation executable. In many cases one would want to filter out this information for analysis of a specific control block(s) or procedure(s). The PM provides a configuration file to select this area of interest. This selected area is referred to as the "viewing window".

Below is a sample configuration file:

```
# Name of the Rebel file of interest
file_name_1.el
{
   # Select everything in function_1()
   _function_1();

   # Select basic block 1 in function_2() only
   _function_2(BB 1);

   # Select basic block 1, hyper block 2, loopbody 3 in function_3()
   _function_3(BB 1, HB 2, LB 3);

   # Selection operation 24 under basic block 1, hyper block 2 and
   # basic block 4 under loopbody 3
   _function_3(BB 1 (OP 24), HB 2, LB 3(BB 4));

   # Selection operation 24 under basic block 1, hyper block 2 and
   # operation 72 under basic block 4 under loopbody 3
   _function_3(BB 1 (OP 24), HB 2, LB 3(BB 4 (OP 72)));
};


# The entire Rebel file is interesting
file_name_2.el{};

# This file too.
file_name_3.el
{
};
```

12. Building performance monitoring tools

For a description of the C++ interface used to build performance monitoring tools, refer to **trimaran/simu/include/Perf/PM_filterator.h**

The PM_filterator class is the heart of this framework. This class extracts out control-flow and/or address events from the execution trace by filtering out events not present in the viewing window.

The event recognized by the PM_filterator class can be found is **trimaran/simu/include/Perf/PM_event.h**

Several example of tools built using this framework have been included with Trimaran. These can be found in **trimaran/simu/src/Perf/Clients**, and include a control flow profiler which annotates control block and edge frequencies onto the control flow graph.

The appendix shows an example of how this profiler is used through a shell.

13. Appendix - Tutorial Example

Here we show a complete example -- generating, compiling and emulating a C simulation file and analyzing the execution (using the above two PM tools).

; Make sure the parameters **address_trace** generation and **control_flow_trace** generation are turned on.
; Use **Trimaran C Compiler** to generate the simulatable executable
; For tcc usage, "tcc -help"
>**tcc** -bench bmm

>ls -l bmm
-rwxrwxr-x trimaran trimaran bmm*

; Create a named pipe DYN_TRACE so simulator emits out trace to a pipe
; instead of a file
>mkfifo DYN_TRACE

; Create a named pipe trace_one
>mkfifo trace_one

; Create a named pipe trace_two
>mkfifo trace_two

; Read contents from DYN_TRACE and broadcast them to t1 and t2
>tee < DYN_TRACE -a trace_one >> trace_two

; Run the control flow profiler
; bmm.cfg has the viewing window configuration file
; Waits for contents to appear on trace_one
>**profiler** bmm.cfg < trace_one &

; Run the address tool
; Waits for contents to appear in trace_two
>**addressTrace** bmm.cfg < trace_two &

; Finally start the simulation
>bmm 20 20 &

; bmm is the trace producer. **profiler** and **addressTrace** are the trace consumers. all three programs run
; simultaneously in the above demonstration!

14. References

1. V. Kathail, M. Schlansker and B. R. Rau. HPL-PD Architecture Specification: Version 1.1. Technical Report HPL-93-80 (R.1). Hewlett-Packard Laboratories, February 1994 (revised July 1999).

2. M. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik and S. G. Abraham. Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity. HPL Technical Report HPL-96-120. Hewlett-Packard Laboratories, February 1997.

3. J. C. Gyllenhaal, W.-m. W. Hwu and B. R. Rau. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3. University of Illinois at Urbana-Champaign, 1996.

4. S. Aditya, V. Kathail and B. R. Rau. Elcor's Machine Description System: Version 3.0. HPL Technical Report HPL-98-128. Hewlett-Packard Laboratories, July 1998.