2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Profiling Virtual Prototypes: Simulation Performance Analysis & Optimization

Sandeep Jain, NXP, Noida, India (sandeep.jain@nxp.com)

***Abstract*—Virtual Prototyping is a proven methodology for early hardware architectural exploration, performance analysis and software development. To meet its objectives, Virtual Prototypes are expected to meet certain requirements around accuracy, early availability and simulation speed. While there has been much research around modeling techniques to help improve simulation performance of Virtual Prototypes, there hasn't been much focus on good profiling tools to measure and analyze simulation performance of Virtual Prototypes. Most off-the-shelf profiling tools, like gprof, oprofile, perf, etc. are primarily designed to profile pre-compiled software code and report generated by these tools is organized for analysis of software function calls. For the purpose of Virtual Prototypes, a much desirable format is a one based on design hierarchy of the SoC being modeled. This paper presents a Virtual Prototype profiler developed to provide a detailed design hierarchy level breakdown of simulation execution time. The profiler is built into the Virtual Prototype and is designed to have negligible overhead.**

Keywords—Virtual Prototype; functional simulator; profiling; simulation performance

I. INTRODUCTION

Virtual Prototyping is a proven methodology for early hardware architectural exploration, performance analysis and software development. To meet its objectives, Virtual Prototypes are expected to meet certain requirements around accuracy, early availability and simulation speed. Modeling Virtual Prototypes to accurately represent actual system being simulated helps increase confidence on reliability of results obtained using Virtual Prototypes. Early availability of Virtual Prototypes to architects and software developers during project phase help increase the utility and impact of Virtual Prototypes on project execution. Besides accuracy and early availability, simulation speed of Virtual Prototype is an equally important aspect – high simulation speed of Virtual Prototypes allows architects and performance analysts to explore large design space in a timely manner as well as helps software developers bring-up full software stack on Virtual Prototype in a relatively short time span. While there has been much research around modeling techniques to help improve simulation performance of Virtual Prototypes, there hasn't been much focus on good profiling tools to measure and analyze simulation performance of Virtual Prototypes. Most off-the-shelf profiling tools, like gprof, oprofile, perf, etc. are primarily designed to profile pre-compiled software code generated by a C/C++ compiler or JVM like code which is JIT compiled but is not updated during the course of application run. Moreover, report generated by these tools is organized for analysis of software function calls, while for the purpose of Virtual Prototypes a much desirable format is a one based on design hierarchy of the SoC being modeled. Off-the-shelf profilers provide information regarding time spent for each function at a software level, however, these profilers don't have any information about design hierarchy being simulated in a Virtual Prototype, so can't provide any visibility into how different component models in a system consume simulation execution time. For example, if a common function is used to implement a component which is instantiated multiple times in Virtual Prototype, off-the-shelf profilers will not be able to distinguish how much time this common function takes in context of different instantiations of same component model.

Due to lack of such a profiler, Virtual Prototype model developers generally rely on analyzing end-to-end simulation time obtained by simulating performance benchmark suites representing typical workloads to be used on Virtual Prototype or a coarse-grained analysis of simulation time spent between simulation kernel and certain component models. This paper presents a Virtual Prototype profiler developed to overcome these limitations to provide a detailed design hierarchy level breakdown of simulation time. Moreover, the profiler is built into the Virtual Prototype and is designed to have negligible overhead. The paper presents the detailed design of the profiler, results obtained by application of this profiler in several full-system Virtual Prototypes and its impact in uncovering several hard-to-find performance issues in simulation. The paper also shows the result on full system simulations with and without enabling the in-built profiler to show that the profiler has negligible overhead as desired.

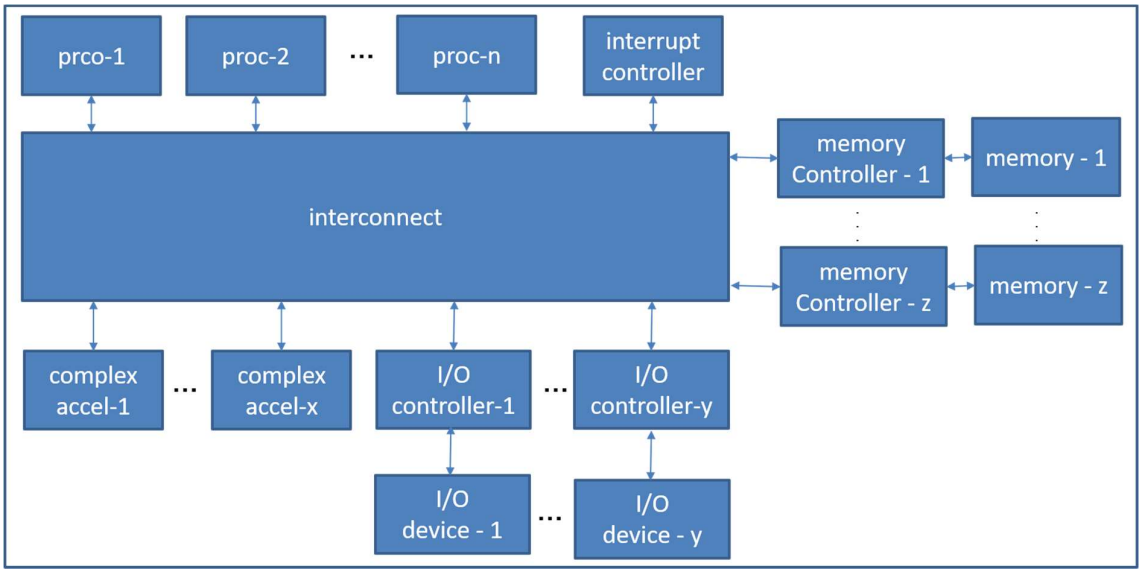
II. RELATED WORK

Much of the prior work listed in references has focused on general techniques for developing high performance software in general and avoiding modeling aspects known to degrade simulation performance. General software techniques include coding styles, in-lining, carefully selecting data structures and containers, avoiding unnecessary logging and I/O, making as fewer copies as possible, etc. Typical modeling techniques recommended for simulation models include avoiding context switches, higher abstraction levels and bursty communication. Typical modeling techniques for instruction set simulators (ISS) include just-in-time (JIT) compilation instead of interpretive execution and direct memory access. Also, much of the prior work focused around applying standard off-the-shelf profilers like gprof, perf, oprofile for Virtual Prototypes which provide only function call level details of time spent during simulation and help in optimizing some of the most-time taking methods to a certain level. However, with regards to Virtual Prototypes, these software function level reports are of limited applicability since it doesn't provide any insight into how various component models are consuming simulation time. Method employed in [2] & [3] report simulation time between simulation kernel vs certain important component models, with the focus on optimizing the kernel. This paper extends the concept to provide a mechanism to get a detailed design hierarchy level breakdown of simulation execution time.

III. INFRASTRUCTURE

The profiler presented in this paper is designed to work with Functional Virtual Prototypes modeled in C/C++ and/or System C and using Transaction Level Models (TLM). Simulation can be controlled via System C kernel or custom C/C++ simulation kernel. A C++ modeling library was developed to abstract much of the complexity involved around simulation kernel and interface semantics and provide simple C/C++ APIs which can be implemented by individual models of the Virtual Prototype. The profiler was developed as part of this modeling library.

Following figure shows a high-level view of a Functional Virtual Prototype. It comprises of many transaction level models of embedded processors, interrupt controller, complex accelerators, abstract interconnect networks, IO peripheral and memory controllers along with associated devices and memory models.



Every component model is classified into two categories in our modeling library, based on whether it needs to be provided simulation time or not by the simulation kernel. All the processor models and some of the complex accelerator models need to be provided simulation time – these are referred to as ***Execution Control*** components in our modeling library. All ***Execution Control*** components work in a co-operative multi-threading fashion where-in each is given a time-slice by the simulation kernel and it yields after performing desired amount of work so that the time-slice can be given to next ***Execution Control*** component by the simulation kernel and the simulation proceeds in this fashion.

Most of the interconnect, IO peripheral and memory controller models along with device and memory models don't need to be provided simulation time as they perform all their work in blocking fashion in context of the ***Execution Control*** components. Such models are referred to as ***Non-Execution Control*** components in our modeling library and are not provided time-slice by the simulation kernel.

The simulation profiler is designed to profile all the component models and provides wall-clock time consumed by each model during simulation. The library to measure the wall clock time consumption should be carefully selected to have negligible performance overhead of its own. The major requirement for this was that the library call should be using Time Stamp Counter (TSC) register of the underlying processor of the host machine on which Virtual Prototype is being run, instead of relying of the system calls of the host operating system. Also, high time resolution should be supported by the library call so that even very minuscule change in wall-clock time can be measured. ***boost::posix_time*** or ***std::chrono*** libraries meet these requirements and can be used. Our framework is independent on the library chosen for wall-clock time measurements and any library of choice can be used. ***get_timestamp()*** is the API which the modeling library expects to be implemented using the library of choice to return the current timestamp on each invocation of this method.

Following was the code snippet for ***Execution Control*** component models for time-slicing without profiler.

```
void do_timeslice(..) { // declared as SC_METHOD
…
do_stuff(..); // implemented by individual component models
…
next_trigger(time_elapsed, sc_core::SC_NS); // dynamic sensitivity (optional)
}
```

To enable profiler, following updates were done to this.

```
void do_timeslice(..) {
…
if (profiling_enabled) now = get_timestamp(); // library call to get current wall clock time stamp
do_stuff(..);
if (profiling_enabled) profile.bucket[idx] += (get_timestamp() – now);
…
next_trigger(time_elapsed, sc_core::SC_NS);
}
```

Wall clock timestamp was measured just before and just after the call to *do_stuff*() method which is implemented by individual component models to perform the desired amount of work for the current time-slice. The delta between the two timestamps provides the wall-clock time consumed by the component model during the current time-slice. The profiler maintains separate wall-clock timestamp buckets (***profile.bucket***) for each component model in Virtual Prototype and keeps incrementing it with the delta for each invocation of time slice for that component.

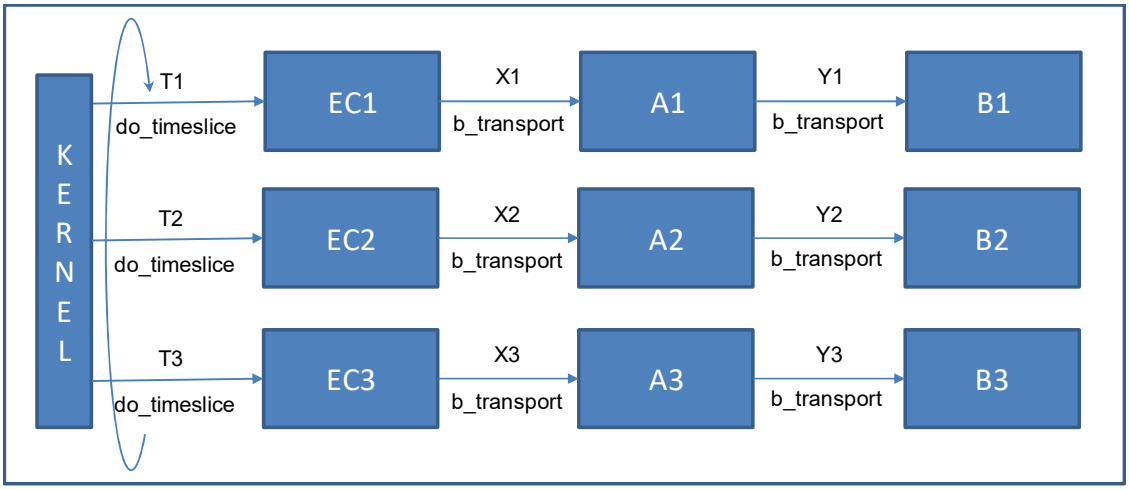
Since ***Non-Execution Control*** components perform their work in context of ***Execution Control*** components, any simulation overhead due to such components gets accumulated with the initiating components. The model-to-model interface methods of the modeling library were updated to measure wall clock time spent outside of the component as shown in the code snippets below. Following was the code snippet from the default interface implementation without profiler.

```
void do_interface_access(..) { // called from do_stuff
…
socket->b_transport(*trans, delay); // implemented by target component model
…
}
```

To enable measurement of wall clock time spent outside of the component, following updates were done.

```
void do_interface_access(..) {
…
if (profiling_enabled) now = get_timestamp();
socket->b_transport(*trans, delay);
if (profiling_enabled) profile.bucket_ext[jdx] += (get_timestamp() – now);
…
}
```

Wall clock timestamp delta before and after the call to blocking transport provides the wall clock execution time spent during the current invocation of blocking transport. The profiler maintains separate external wall clock time buckets (***profile.bucket_ext***) for each component model in Virtual Prototype and keeps accumulating the time delta for each invocation of blocking transport call from that component. The difference between the ***profile.bucket*** and ***profile.bucket_ext*** provides the wall clock time spent during execution of the component itself. This is elaborated using following figure.



Simulation Kernel provides time-slice to each Execution Control component (EC1, EC2, EC3) and wall clock timestamp measurements are done for each of these which includes the overall execution time of the ***Execution Control*** component and all ***Non-Execution Control*** components (A1, A2, A3, B1, B2, B3) called in the context of the current ***Execution Control*** component during the current time-slice. Let these be T1, T2, T3, etc. as shown in the figure. Let the wall clock timestamp measurements at the interface levels be X1, X2, X3, Y1, Y2, Y3, etc. as shown in the figure. Then the following formula can be applied to get the execution timestamp of each component for the current time-slice:

```
EC1 execution time: T1 - X1
EC2 execution time: T2 - X2
EC3 execution time: T3 – X3
A1 execution time: X1 - Y1
A2 execution time: X2 - Y2
A3 execution time: X3 – Y3
        B1 execution time: Y1
B2 execution time: Y2
B3 execution time: Y3
…
```

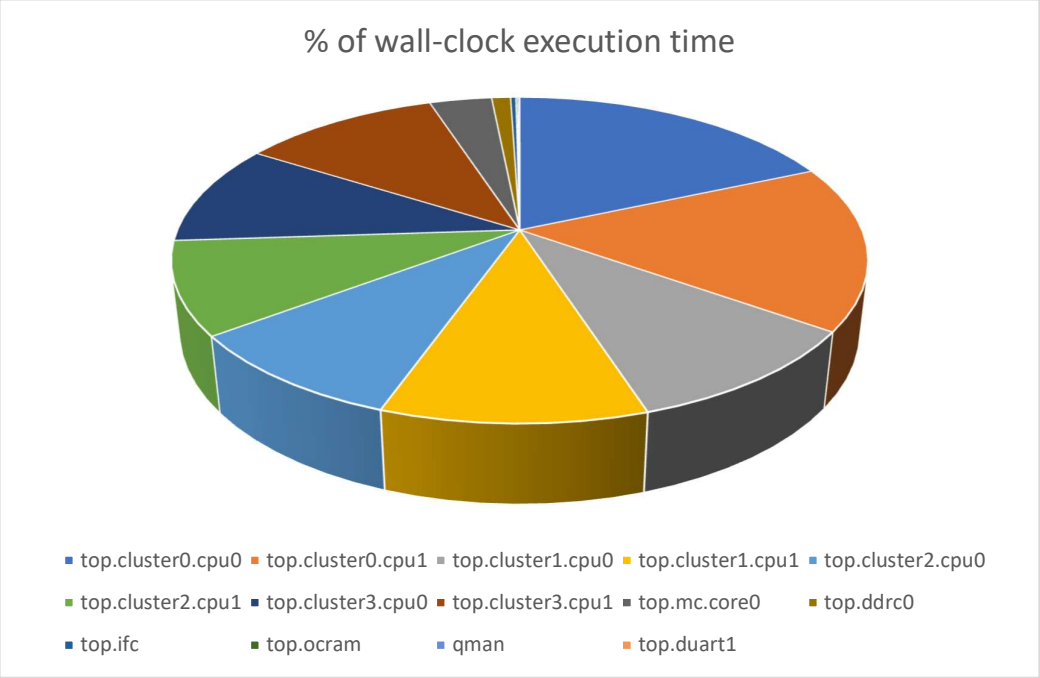
Wall clock execution time for each component in the Virtual Prototype is logged at the end-of-simulation. A debug callback to get this information dynamically at run-time is also supported.

IV. RESULTS

Virtual prototype profiler presented in this paper can be applied to any Virtual Prototype development methodology and helps in gaining useful information about simulation execution time spent among various components at a design hierarchy level. This helps gaining valuable insight into performance of different component models and helps optimizing the Virtual Prototype to achieve maximum simulation speed. Moreover, since the profiler is built into the Virtual Prototype and can be enabled/disabled at run time, it is useful in measuring the impact of integrating any new component model into the Virtual Prototype, impact of integrating any third-party model on overall simulation performance of the Virtual Prototype as well as how different software and run-time configurations impact simulation performance of the Virtual Prototype.

Following is an example of the report generated by the profiler for 8-core benchmark

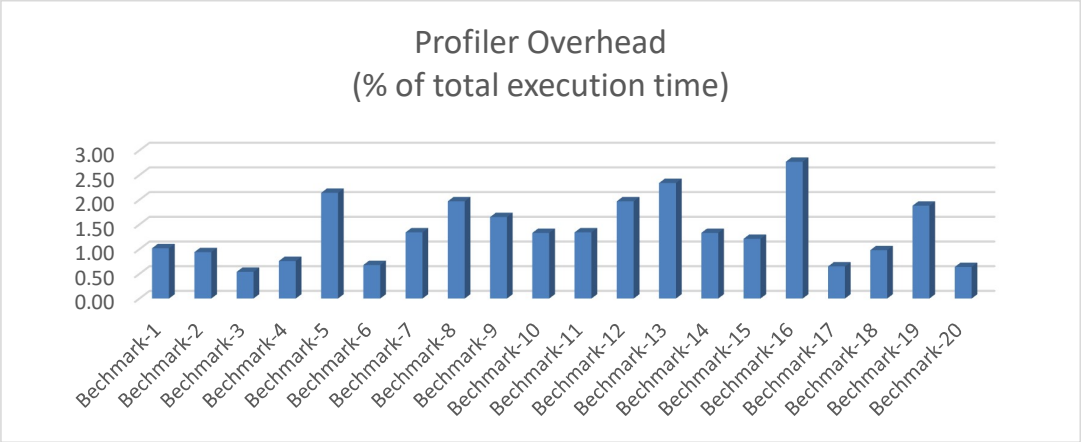
Component	% of wall-clock execution time
top.cluster0.cpu0	18.33
top.cluster0.cpu1	16.32
top.cluster1.cpu0	10.13
top.cluster1.cpu1	10.36
top.cluster2.cpu0	9.24
top.cluster2.cpu1	9.17
top.cluster3.cpu0	9.99
top.cluster3.cpu1	10.95
top.mc.core0	3.43
top.ddrc0	1.04
top.ifc	0.29
top.ocram	0.10
qman	0.10
top.duart1	0.02



For one of the software workloads, the profiler helped uncover that one of the UART model instances was consuming most simulation time, which was both undesirable and unexpected. But having this uncovered, it was simple to fix the issue by removing the unnecessary synchronization requirement enforced by the model and help improve simulation performance by around 200%.

In another scenario, one of the CPU model instances was found to take significantly higher simulation time that other CPU model instances in an SMP system. It turned out to be a run-time configuration issue of the Virtual Prototype which caused that CPU model instance to get stuck in an infinite loop of exceptions and hence significantly degrading simulation performance of the overall Virtual Prototype. Fixing it helped improve simulation performance by around 800%.

Following chart shows the profiler overhead over a range of 20 benchmarks comprising of SMP Linux boot, IP-Fwd, IP-Sec, EEMBC and SPEC.



The measured overhead of the in-built profiler across a range of 20 benchmarks was less than 1.5% on an average and approx. 2.8% in the worst case.

V. CONCLUSION

The Virtual prototype profiler presented in this paper was applied to several Virtual Prototypes developed for both architectural exploration and software development. The profiler was enabled at run-time to measure impact of different types of software applications and different run-time configuration settings of Virtual Prototypes on simulation performance of the different component models. This helped in uncovering and fixing several performance issues in simulation models as well as figuring out the best possible configurations of various Virtual Prototype run-time configuration options to achieve maximum possible simulation speed for different types of software workloads. The paper provided details of some of the examples where the profiler helped improve simulation performance significantly. Lastly, but not the least, the paper shows simulation results obtained across a variety of software workloads and benchmarks to showcase that the in-built profiler has a negligible overhead.

REFERENCES

[1] Joseph Chapman, “Practical Techniques for Improving SystemC Simulation Performance”, NASCUG 2007

[2] Denis Becker, Matthieu Moy, Jerome Cornet, “SyncView: Visualize and Profile SystemC Simulations”, Design Automation for Understanding Hardware Designs, 2016

[3] Liana Duenha, Rodolfo Azevedo, “Profiling High Level Abstraction Simulators of Multiprocessor Systems”, WCAS, 2012

[4] Dr. Greg Tumbush, “Dramatically Increase the Performance of SystemC Simulations”, DVCon, 2007

[5] Rocco Jonack, “VP Performance Optimization – How to analyze and optimize the speed of SystemC TLM models”, DVCon Europe, 2014