Profiling Virtual Prototypes: Simulation Performance Analysis & Optimization

Sandeep Jain, NXP, Noida
Agenda

• Motivation
• Requirements
• Modeling Library
• Wall-clock time measurement
• Profiler Design
• Results
• Conclusion
Motivation

• Virtual Prototyping is a proven methodology for early HW architectural exploration, performance analysis and software development

• Lots of guidelines available for modeling techniques to improve simulation performance of Virtual Prototypes
  – Minimize Context switching (minimize usage of SC_THREAD)
  – Make as fewer copies as possible (use pointers instead of data copies)
  – Use higher level of abstraction (model only what is required)
  – Use JIT-ISS instead of interpretive-ISS
  – Prefer DMI over blocking transport

• There is lack of profiling tools geared towards VP
Motivation

• Most off-the-shelf profilers like gprof, oprofile, perf, etc. are primarily designed to profile code for analysis of software function calls
  – These profilers lack visibility into the design hierarchy of the Virtual Prototype
  – For any method used to implement a common function across different component models, these profilers can’t distinguish how much time it takes in context of these different models

• For profiling VP, a desirable format is one based on design hierarchy of system being modeled
Requirements

• Simulation execution time consumed by various component models as per the design hierarchy modeled by VP
• Negligible overhead
  – Enabling the profiler should have minimal impact of overall simulation performance
• Non intrusive
  – No change required to the component models to enable profiling
• No re-compilation to enable profiler
VP comprises of many TLM of processors, accelerators, interconnects, IO peripheral and memory controllers and associated device and memory models.
Modeling Library

• Transaction Level models (System C, C, C++)
• System C kernel or custom simulation kernel in C/C++
• In-house Modeling library to expose simple C/C++ APIs to model developers
  – Hide most of the simulation kernel and interface semantics and associated complexities
  – Uniform coding style across all models
• **Execution Control** vs **Non-Execution Control** component models
  – Execution Control component models are provided simulation time by the kernel
  – Co-operative multi-threading
  – Non-Execution Control component models execute in context of Execution Control models in a blocking fashion
Wall-clock Time Measurement

• Requirements
  – Negligible overhead
    • Use Timestamp Counter (TSC) register of the underlying processor of the host machine
    • Avoid relying on system-call provided by host operating system
  – High resolution
    • Able to measure miniscule changes in time
• boost::posix_time or std::chrono found to be good candidates
• Profiling Framework is independent of the library chosen
  – \texttt{get_timestamp()} is the API used by profiler which can be implemented using the library of choice
Profiler Design

• Execution Control components code snippet without profiler

```c
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)
}
```

• Execution Control components code snippet with profiler

```c
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);
}
```

• Separate buckets in profiler for maintaining execution time consumed by each Execution Control
• Since Non-Execution Control component models work in context of Execution Control models, any overhead by these get accumulated into the corresponding Execution control component model
Profiler Design

• Component model interface code snippet without profiler

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

• Component model interface code snippet with profiler

```c
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);
    ...
}
```

• Separate buckets in profiler for maintaining execution time consumed outside of each component model

• Difference between the `profile.bucket` and `profile.bucket_ext` provides the wall clock time spent during execution of the component itself
Profiler Design

EC1 execution time: $T_1 - X_1$
EC2 execution time: $T_2 - X_2$
EC3 execution time: $T_3 - X_3$
A1 execution time: $X_1 - Y_1$
A2 execution time: $X_2 - Y_2$
A3 execution time: $X_3 - Y_3$
B1 execution time: $Y_1$
B2 execution time: $Y_2$
B3 execution time: $Y_3$

...
Results

- Following is a sample report from the profiler for a workload with 8-cores

<table>
<thead>
<tr>
<th>Component</th>
<th>% of wall-clock execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>top.cluster0.cpu0</td>
<td>18.33</td>
</tr>
<tr>
<td>top.cluster0.cpu1</td>
<td>16.32</td>
</tr>
<tr>
<td>top.cluster1.cpu0</td>
<td>10.13</td>
</tr>
<tr>
<td>top.cluster1.cpu1</td>
<td>10.36</td>
</tr>
<tr>
<td>top.cluster2.cpu0</td>
<td>9.24</td>
</tr>
<tr>
<td>top.cluster2.cpu1</td>
<td>9.17</td>
</tr>
<tr>
<td>top.cluster3.cpu0</td>
<td>9.99</td>
</tr>
<tr>
<td>top.cluster3.cpu1</td>
<td>10.95</td>
</tr>
<tr>
<td>top.mc.core0</td>
<td>3.43</td>
</tr>
<tr>
<td>top.ddrc0</td>
<td>1.04</td>
</tr>
<tr>
<td>top.ifc</td>
<td>0.29</td>
</tr>
<tr>
<td>top.ocram</td>
<td>0.10</td>
</tr>
<tr>
<td>qman</td>
<td>0.10</td>
</tr>
<tr>
<td>top.duart1</td>
<td>0.02</td>
</tr>
</tbody>
</table>
Results

• Following chart shows the profiler overhead over a range of 20 benchmarks comprising of SMP Linux boot, IP-Fwd, IP-Sec, EEMBC, SPEC
• Average overhead was less than 1.5% with approx. 2.8% for worst case
Conclusion

• VP profiler was applied to several VP for various SOCs for early software bring-up
• VP profiler provided simulation execution time consumed by various component models as per the design hierarchy modeled by VP with negligible overhead on simulation performance
• VP profiler didn’t require any changes in individual component models
  – It was fully implemented within the modeling library abstraction layer
• VP profiler can be enabled at run-time to measure impact of different software applications and different VP configuration options on simulation performance of different component models
• VP profiler helped uncover several performance issues in simulation models as well as optimal configuration options to be achieve best simulation performance
Questions