A Complete SystemC Process Instrumentation Interface and Its Application to Simulation Performance Analysis.

Bishnupriya Bhattacharya[1], Chandra Sekhar Katuri[1], Vincent Motel[2]

[1] Cadence Design Systems, RMZ Ecospace, Sarjapur Outer Ring Road, Bangalore 560103 India
[2] Cadence Design Systems, 29 boulevard des Alpes, 38246 Meylan, France
{bpriya, csekhar, vmotel}@cadence.com

***Abstract*- In the Electronic System Level domain, the SystemC language uses non-preemptive processes to simulate hardware parallelism in C++ [1]. Processes are a fundamental construct of the language, which makes modeling of hardware easier than in plain C++. Since processes are a key design element of SystemC, there are various application domains (e.g. debugging, performance analysis) where a generic mechanism to associate user-defined callbacks with process state changes will prove to be beneficial. We propose such a generic interface for registering user-defined callbacks triggered at SystemC process state changes. We also present an application case study for such an interface for simulation performance profiling in a virtual platform and in an approximately-timed memory controller model.**

I. Introduction

Hardware description languages, e.g. SystemVerilog [2], VHDL [3] use processes to model the parallel nature of electronic systems and to simulate the concurrent behavior of multiple elements, even if the simulation execution is sequential.

In the Electronic System Level domain, the SystemC language [1] uses non-preemptive processes to simulate hardware parallelism. SystemC is designed as a C++ class library [4], and processes are a fundamental construct of the SystemC language that makes modeling of hardware easier than in plain C++. There are two kinds of SystemC processes: methods and threads. A method process executes completely from beginning to end in each execution, and does not maintain any local state. A thread process maintains its own local state, and on execution can voluntarily suspend itself using the *wait()* construct. SystemC defines a co-routine semantics [1] of process execution. When a process is created, it can be specified to trigger on specific events (e.g. signal value change). This is called the static sensitivity of that process. When a process executes, it can dynamically change its sensitivity to a different set of event triggers. When a process's sensitivity triggers, the process is scheduled for execution. A method process always executes from beginning to end, and returns control back to the SystemC scheduler. Once a thread process starts execution, it voluntarily yields either to the next thread process, or to the SystemC scheduler, or terminates itself by exiting its associated function. The SystemC scheduler never preempts a process forcefully. In addition, SystemC also defines process control constructs using which a process can explicitly suspend, resume, disable, enable, kill, reset or throw an exception in another process.

Given that processes are the fundamental behavioral block in SystemC, there are various application domains (e.g. debugging, performance analysis) where a generic mechanism to associate user-defined callbacks with process state changes will prove to be beneficial. In this paper, we propose such a language extension to register user-defined callbacks triggered at process state changes including creation, activation, suspension, resumption, termination, reset, disable, enable, etc.

The rest of the paper is organized as follows: In Section II, we present our process instrumentation interface, differentiating with previous work in this area. In Section III, we discuss implementation of this interface, keeping performance concerns in mind. In Section IV, we present an application of our process instrumentation interface for simulation performance profiling in a virtual platform and in an approximately-timed memory controller design, and we analyze the results. Finally, we conclude in Section V with some directions for future work.

II. SYSTEMC PROCESS INSTRUMENTATION INTERFACE

A. *Previous Work*

An initial proposal for process callbacks (and event callbacks) was submitted as a part of the P1666-2011 standardization process [5], but did not make it to the standard. Our work takes into account the proposal in [5], along with the feedback provided in the Accellera SystemC Language Working Group [6], and proposes significant enhancements and modifications to come up with a complete and comprehensive solution for registering process state change callbacks.

B. *Requirements for a complete process instrumentation interface*

The main additional requirements over [5] were:

- Method and thread processes should be handled consistently, for ease of use
- All process control constructs should also be uniformly handled
- The triggering signal for a process reset should be added; from user experience, this is a useful piece of information
- Details of process creation callback should be specified, with hook to *sc_spawn_options* class for completeness
- Language-standard *sc_process_handle* class should be used instead of implementation-defined class *sc_process_b*, for 'safer' callbacks
- Callback registration should be available at different granularity levels ranging from individual process handles to all processes in a module hierarchy to all processes in a design
- Most importantly, rigorous details of the semantics should be specified

C. *Specification of the process instrumentation interface*

The *sc_process_callback* class defines the process callback interface. The user derives a custom class from the *sc_process_callback* base class and registers an instance of the custom class using the global callback registration functions. A SystemC simulator invokes a callback when the process associated with the callback changes its state.

Class Definition of the sc_process_callback Base Class

```
class sc_process_callback {
public:
   /* Called after the method or thread is created */
   virtual void process_created(sc_process_handle process){}
   /* Called before invoking a method or thread */
   virtual void process_activated(sc_process_handle process){}
   /* Called after halting/suspending a method or thread */
   virtual void process_halted(sc_process_handle process){}
   /* Called after a method or thread is terminated */
   virtual void process_terminated(sc_process_handle process){}
   enum process_control_construct_t {
      SUSPEND, RESUME, DISABLE, ENABLE,
      KILL, RESET,
      SYNC_RESET_ON, SYNC_RESET_OFF,
      RESET_SIGNAL_IS, ASYNC_RESET_SIGNAL_IS
```

```
    };
    /* Called before invoking a process control construct*/
    virtual void process_control_construct_invoked(
      sc_process_handle process,
      process_control_construct_t kind
    );
    /* Called when port/signal specified in {async_}reset_signal_is() changes value */
    enum reset_active_t {ACTIVE, INACTIVE};
    virtual void reset_signal_changed(
      sc_process_handle process,
      const char* reset_sig_name,
      reset_active_t reset_sig_state
    );
    virtual ~sc_process_callback() {}
private:
    sc_process_callback() {}
};
```

The member methods of *sc_process_callback* class have the following semantics:

- *process_created()*: This callback is invoked when the process is created. Processes can be created statically or dynamically using *sc_spawn()*. For both cases, the *process_created()* callback is invoked as soon as the process is created.
- *process_activated()*: This callback is invoked before the process is activated and every time control goes to a function associated with a process. For methods, this callback is called every time a method executes from the beginning. For threads, it is called when a thread is first started and every time the thread resumes. Note that the *process_activated()* callback is not called exactly before executing the first line or the resumed line of the associated function. It gets invoked before the function executes, but some implementation code executes in between the callback being called and the function code executing. Exactly how much implementation code executes is implementation dependent. An implementation must guarantee that the same amount of code executes uniformly for every thread process and for every method process.
- *process_halted()*: This callback is invoked when a process halts. For a method process, it is called when a method completes. For a thread process, it is called whenever the process suspends itself. This callback is called close to the point after a process suspends, and is uniform for all threads and methods. The exact quantification is implementation-defined.
- *process_terminated()*: For a thread, this callback is invoked when a thread returns from its associated function. This can be voluntary or because of a process control construct such as *kill()*, *reset()*, or *throw_it()*. For a method, this callback is invoked only if a method is explicitly killed using the process control construct *kill()*. Similar to the other callbacks, exact timing of this call is implementation-dependent but close to the actual termination, and consistent across threads and methods.
- *process_control_construct_invoked()*: This callback is invoked when a process control construct is issued on the process handle. The exact process control construct that was issued is provided in the 'reason' argument of type *process_control_construct_t*.
- *reset_signal_changed()*: This callback is invoked when the port/signal specified in *reset_signal_is()* or *async_reset_signal_is()* changes value. The additional parameters to this callback specify if the reset signal/port went active or inactive, and the name of the reset signal/port.

Callback Registration Functions:

Callback registration for all processes in a design

```
bool sc_add_process_callback_all(
   sc_process_callback    *callback
);
bool sc_remove_process_callback_all(
   sc_process_callback *callback
);
```

The *sc_add_process_callback_all()* global function registers process callbacks for all processes in a design, including the processes that exist at the time of the registration, and processes that are created later. For processes that already exist when the registration function is called, the *process_created()* callback is missed, and possibly other state change callbacks are also missed that have already happened before. For processes that are created after the registration function is invoked, every state change is triggered, including the *process_created()* callback.

When invoking a callback registered with the *sc_add_process_callback_all()* function for a process instance, the implementation creates a local *sc_process_handle* for that process instance and calls the process callback with that local process handle. This local process handle goes out of scope once the process callback returns.

It is legal to call the *sc_add_proces_callback_all()* registration function from a static initializer.

If the same *sc_process_callback* instance is registered multiple times using the *sc_add_process_callback_all()* function, only the first registration applies and subsequent ones are ignored with a warning. If there are multiple invocations of *sc_add_process_callback_all()* function, multiple process callbacks are registered with all the processes.

The sc*_remove_process_callback_all()* function de-registers any callbacks previously registered with the *sc_add_process_callback_all()* function. An error is raised if the *sc_remove_process_callback_all()* function is invoked with a *sc_process_callback* instance that was not previously registered using function *sc_add_process_callback_all()* function.

The *sc_add_process_callback_all()* and *sc_remove_process_callback_all()* functions return true for successful registration or de-registration, and false otherwise.

Callback registration for all processes in a module

```
bool sc_add_process_callback_module(
   sc_module* mod, sc_process_callback *callback
);
bool sc_remove_process_callback_module(
   sc_module* mod, sc_process_callback *callback
);
```

The global *sc_add_process_callback_module()* and *sc_remove_process_callback_module()* functions register and de-register process callbacks for all processes that belong to the hierarchy of a given module. The first argument to these functions is the module pointer, and it is the user's responsibility to ensure the module pointer is valid when these functions are invoked.

If the module registration function is called after processes inside the module hierarchy are already created, all such processes in that module's hierarchy will miss the *process_created()* callback.

If during simulation, a new process is created using the *sc_spawn()* routine inside a module's hierarchy, the specified *sc_process_callback* class instance gets registered with the newly created process. The *process_created()* callback is called on the newly created process instance before the *sc_spawn()* call returns.

When invoking a callback registered with *sc_add_process_callback_module()* function for a process instance, the implementation creates a local *sc_process_handle* object for that process instance, and calls the process callback with that local process handle. This local process handle goes out of scope once the process callback returns.

If the same *sc_process_callback* instance is registered multiple times on the same module using the *sc_add_process_callback_module()*, then only the first registration applies and subsequent registrations are ignored with a warning.

An error is raised if the *sc_remove_process_callback_module()* is invoked with a *sc_process_callback* instance that was not previously registered using the *sc_add_process_callback_module()* function on the same module instance.

The *sc_add_process_callback_module()* and *sc_remove_process_callback_module()* functions return the status as true for successful registration or de-registration, and false otherwise.

Callback registration for a specific process in a design

```
bool sc_add_process_callback(sc_process_handle process,
                             sc_process_callback *callback,
                             sc_descendant_inclusion_info
                                include_descendants =
                                SC_NO_DESCENDANTS
);
bool sc_remove_process_callback(sc_process_handle process,
                                sc_process_callback *callback,
                                sc_descendant_inclusion_info
                                   include_descendants =
                                   SC_NO_DESCENDANTS
);
class sc_spawn_options {
 public:
    ...
    bool add_process_callback(sc_process_callback *);
};
```

Given a process handle as first argument, the global functions *sc_add_process_callback()* and *sc_remove_process_callback()* register and deregister process callbacks respectively, for that process handle. The third parameter to these functions indicate if the registration applies to the entire process hierarchy tree rooted at the *sc_process_handle*, including future descendants created using *sc_spawn()*.

If the *sc_process_handle* is invalid, a warning is generated and the registration or deregistration functions have no effect.

If same process callback instance is registered to same process handle multiple times, a warning is generated for the subsequent calls, and only the first registration applies.

These functions return true for successful registration or deregistration, and false otherwise.

The function *sc_spawn()* returns a *sc_process_handle*. This handle can be used with registration function *sc_add_process_callback()* to register process callbacks. However, when *sc_spawn()* is called during simulation, by the time *sc_spawn()* returns, the process has already been created, so the *process_created()* callback is missed. Hence, for completeness, an additional interface to register process callbacks is provided for *sc_spawn_options*. The behavior of *sc_spawn_options::add_process_callback()* is to attach the callbacks implicitly to the *sc_process_handle* returned from the *sc_spawn()* call to which the *sc_spawn_options* instance is passed as an argument. In that case, the *process_created()* callback fires before the *sc_spawn()* call returns.

Callback Registration is Cumulative:

A single process instance can have multiple callbacks registered by using different kinds of registration functions. For example, a global callback can be registered using the *sc_add_process_callback_all()* function. There can also be module-specific callbacks registered on parent and child modules *top* and *top.mid* using the *sc_add_process_callback_module()*. The process *top.mid.p1* will then have the following callbacks registered.

- Global callbacks
- Callbacks registered with *top* module
- Callbacks registered with *top.mid* module

The order in which these callbacks are invoked is implementation-defined.

Callback Rules:

- Multiple callbacks can be attached to the same process.
    - The order in which the callbacks are triggered is implementation defined.
- Same callback object can be attached to multiple processes. Callbacks are invoked with the change in process state with the respective process handle.
    - User manages the lifetime of the callback object. Process callback needs to be removed from registration before it is deleted.
- Exceptions
    - All exceptions raised from process callbacks including the SC_REPORT shall be caught and re-thrown.
- Callbacks cannot consume time
    - Callback cannot consume time (‘*wait()*’ call is not allowed).
- Callbacks cannot call process control constructs.

III. Implementation of the Process Instrumentation Interface

A subset of the process callback interface specified in Section II, has been implemented in the Cadence Incisive® simulator, in the 15.10 release. Callback functions *process_control_construct_invoked()* and *reset_signal_changed()* gives more information about the callback state and are useful for advanced use case. These callback implementations are planned for future releases. Callback registration for individual process handle and *sc_spawn_options::add_process_callback()* also qualify as finer-grained, advanced use models that have been left for future implementation.

The implementation requires fundamental kernel level instrumentation that affects scheduling. One key concern when implementing such a kernel-level instrumentation is performance for the common case. Our implementation has been carefully fine-tuned, and demonstrates no performance degradation in the common case when process callbacks are not registered.

IV. Applications of the Process Instrumentation Interface

A. Overview of possible applications

Access to process activity details should enable many applications, mainly in the debugging domain (recording and visualization of process execution sequence) and in the performance analysis domain.

An application may collect only statistics on process execution, without recording the details. Using the process callback interface, it is very simple to make counts such as the number of activations of each process.

The application may also keep a record of the process activations in a database. That information can be enriched with SystemC simulation time, which allows display in a waveform viewer along with other simulation traces.

Knowing the details of process activity can help to answer very specific questions too. For example in [7], when the authors needed to measure the number of runnable process of each delta cycle to evaluate parallelization possibilities, they had to modify the ASI implementation of the SystemC kernel. By using the SystemC process instrumentation interface, it would be possible to count the number of processes actually executed at each delta cycle without any modification of the kernel, which gives an upper bound of the number of runnable process.

B. Application to simulation performance analysis

We have used the process instrumentation interface to make accurate simulation performance measurements. An example application monitors the host CPU time usage of each process activation: it records the current CPU time usage of the simulation from the *process_activated()* callback, and records it again from the *process_halted()* or *process_terminated()* callbacks. The difference corresponds to the amount of CPU time used by the process. As we want to analyze the complete design, we have used the function *sc_add_process_callback_all()* to register these callbacks for all processes in the design.

On the Linux host platform, we have computed the CPU time measurement using the function *clock_gettime()* of *time.h*, with clock id set to *CLOCK_PROCESS_CPUTIME_ID*. This clock setting is very appropriate for simulation performance analysis because it counts only the CPU time used by the simulation, which makes the time measurement independent of the load of the host CPU (unlike measurements based on elapsed time). In addition, it has a nanosecond resolution. Although its actual accuracy is certainly not at that level and probably closer to the microsecond, it is better than a clock at microsecond resolution, because it avoids the abrupt accuracy loss for execution times close to the microsecond, which is quite common for many process executions in typical simulations.

The CPU time usage can be accumulated in simple per-process statistical bins, so that at the end of the simulation, the total CPU time usage of each process instance can be reported. In combination with the process activation counts described above, the average CPU time of the activation of each process can be computed directly. By adding a few variables for each process instance, min and max CPU time can be reported too.

C. Case studies

We applied our performance analysis application to two designs: a complete virtual platform running an OS, and the approximately-timed model of a memory controller.

The virtual platform [8] is a model of the Xilinx Zynq®-7000 “All Programmable SoC” [9], containing an ARM FastModel of the main processor (dual core ARM® Cortex™-A9) and many peripherals, modeled according to the loosely-timed coding style [1]. The platform first boots the Linux OS, and then launches an application processing an MRI image, using the model of a hardware accelerator.

The report generated from our performance analysis tool is shown in Fig. 1. We can see that the SystemC process corresponding to the processor model largely dominates the activity of the virtual platform. This is often observed when the TLM [1] peripherals are highly optimized. Note that in case of a performance bug, such a bug can be easily detected in our report by too many activations of a process, or by unexpectedly large average time of each activation (denoting non-optimal design). In this simulation, the time spent in the hardware accelerator model is small, accounting for approximately 0.2% of the total CPU usage.

The second example is a four-way memory controller model used for system architecture exploration, enabling estimation of system performance metrics such as throughput and latency. The multiple stages of the architecture (FIFOs, arbitration, reordering queue) are modeled in TLM approximately-timed coding style, using the non-blocking transport interface [1]. Incoming traffic is created by four initiator models using SCV constrained randomization [10], and these processes are called “TOP.im_*N*.generate_traffic” in the report.

We can see in Fig. 2 that in this design, the execution time is more balanced between the SystemC processes. Also the processes have quite different average activation times, depending on the amount of computation in each process. The modeling engineer can evaluate for each process if it is expected or not, depending on the actual code content of the process.

```
-- summary
  total CPU usage        =   279.423191s
  total non-process time =     5.329797s
  total process time     =   274.093394s
  number of process activations = 1516532, average 180.736967 us/activation
--list of contributions
    1.90743 %     5.329797 s                         3.514 us/activation - total non-process time (kernel, profiling itself)
   -- processes sorted by total exec time --
   95.26493 %   266.192300 s  517404 activations    514.477 us/activation - zynq.zynq_ps.armfm_core.ARMCortexA9MPx2CT32.run
    1.99865 %     5.584694 s   12000 activations    465.391 us/activation - zynq.ui.run
    0.28222 %     0.788582 s  859786 activations      0.917 us/activation - zynq.zynq_ps.usb1.start_schedule_th
    0.20792 %     0.580972 s    1281 activations    453.530 us/activation - zynq.bp.run
    0.11588 %     0.323805 s       2 activations 161902.519 us/activation - zynq.term_1.xterm_thread
    0.07198 %     0.201141 s   68698 activations      2.928 us/activation - zynq.term_0.xterm_thread
    0.05826 %     0.162791 s       1 activation  162790.992 us/activation - zynq.ddr_ram.reset_process
    0.03854 %     0.107692 s       3 activations  35897.204 us/activation - zynq.loader.reset_thread
    0.02024 %     0.056542 s   14120 activations      4.004 us/activation - zynq.term_0.rx_fifo_method
    0.00714 %     0.019961 s    2400 activations      8.317 us/activation - zynq.fb_ram.run
    0.00567 %     0.015841 s   17395 activations      0.911 us/activation - zynq.zynq_ps.armfm_core.run_standbywfi0
    0.00531 %     0.014839 s    4819 activations      3.079 us/activation - zynq.term_1.rx_fifo_method
    0.00349 %     0.009750 s    1112 activations      8.768 us/activation - zynq.zynq_ps.slirp_wrapper.poll_slirp
    0.00316 %     0.008818 s   11414 activations      0.773 us/activation - zynq.zynq_ps.armfm_core.run_standbywfi1
    0.00163 %     0.004544 s    1133 activations      4.011 us/activation - zynq.zynq_ps.armfm_core.bridge_ints27.signal
    0.00141 %     0.003935 s       1 activation    3934.544 us/activation - zynq.smc_sram_1.reset_process
    0.00140 %     0.003910 s       1 activation    3910.188 us/activation - zynq.smc_sram_0.reset_process
    0.00082 %     0.002304 s     595 activations      3.872 us/activation - zynq.zynq_ps.armfm_core.bridge_ints44.signal
    0.00078 %     0.002178 s       1 activation    2177.840 us/activation - zynq.logiCVCML.run
    0.00046 %     0.001298 s    1133 activations      1.146 us/activation - zynq.zynq_ps.armfm_core__ints27_s2w_convert
```

Figure 1. Performance report on the Zynq Virtual platform (top 20 processes).
SystemC process corresponding to the processor model is highlighted in blue, and the hardware accelerator process is in green.

```
-- summary
  total CPU usage        =    19.445972s
  total non-process time =     3.196789s
  total process time     =    16.249182s
  number of process activations = 3372820, average 4.817684 us/activation
--list of contributions
   16.43934 %     3.196789 s                         0.948 us/activation - total non-process time (kernel, profiling itself)
   -- processes sorted by total exec time --
   39.31143 %     7.644490 s  259107 activations     29.503 us/activation - TOP.mem_controller_0.smart_mem_q.send_transaction
   10.11653 %     1.967258 s  178498 activations     11.021 us/activation - TOP.im_2.generate_traffic
    9.33906 %     1.816071 s  171553 activations     10.586 us/activation - TOP.im_0.generate_traffic
    5.33261 %     1.036977 s   96177 activations     10.782 us/activation - TOP.im_3.generate_traffic
    2.66017 %     0.517297 s  242127 activations      2.136 us/activation - TOP.mem_controller_0.r_arb.transmit_responses
    2.17587 %     0.423119 s  501250 activations      0.844 us/activation - TOP.mem_controller_0.smart_mem_q.accept_requests
    1.97593 %     0.384239 s   38695 activations      9.930 us/activation - TOP.im_1.generate_traffic
    1.97324 %     0.383717 s  585061 activations      0.656 us/activation - TOP.mem_controller_0.r_arb.arbitrate_and send_trans
    1.93752 %     0.376769 s   89170 activations      4.225 us/activation - TOP.mem_controller_0.fifo_2.transmit_responses
    1.88048 %     0.365678 s   85658 activations      4.269 us/activation - TOP.mem_controller_0.fifo_0.transmit_responses
    1.06135 %     0.206390 s   48073 activations      4.293 us/activation - TOP.mem_controller_0.fifo_3.transmit_responses
    1.00635 %     0.195694 s  242126 activations      0.808 us/activation - TOP.memory_0.end_of_busy_state_method
    0.83354 %     0.162090 s  126331 activations      1.283 us/activation - TOP.mem_controller_0.fifo_2.send_requests
    0.81893 %     0.159249 s  120481 activations      1.322 us/activation - TOP.mem_controller_0.fifo_0.send_requests
    0.77634 %     0.150966 s  179744 activations      0.840 us/activation - TOP.mem_controller_0.fifo_2.accept_requests
    0.74205 %     0.144300 s  171597 activations      0.841 us/activation - TOP.mem_controller_0.fifo_0.accept_requests
    0.43911 %     0.085389 s   58443 activations      1.461 us/activation - TOP.mem_controller_0.fifo_3.send_requests
    0.43163 %     0.083935 s   97895 activations      0.857 us/activation - TOP.mem_controller_0.fifo_3.accept_requests
    0.41969 %     0.081612 s   19229 activations      4.244 us/activation - TOP.mem_controller_0.fifo_1.transmit_responses
    0.16556 %     0.032194 s   22471 activations      1.433 us/activation - TOP.mem_controller_0.fifo_1.send_requests
    0.16043 %     0.031198 s   38457 activations      0.811 us/activation - TOP.mem_controller_0.fifo_1.accept_requests
    0.00128 %     0.000248 s     241 activations      1.031 us/activation - TOP.im_1.sync_process
    0.00086 %     0.000168 s     241 activations      0.696 us/activation - TOP.im_0.sync_process
    0.00058 %     0.000113 s     161 activations      0.701 us/activation - TOP.im_2.sync_process
    0.00011 %     0.000021 s      34 activations      0.627 us/activation - TOP.im_3.sync_process
```

Figure 2. Performance report on the memory controller model (complete).

D. *Comparison to sample-based profiling*

Cadence Incisive Simulator already provides a way to measure time spent in each process, by using statistical sample-based profiling, combined with SystemC awareness.

The sample-based profiling of Incisive has numerous advantages: it can support multiple HDL languages, as well as mixed-languages simulations, and it is tightly integrated in the simulator, with a TCL control for ease-of-use and flexibility. Also its impact on performance is low and regular. But to provide accurate results, that method requires a high number of samples, so that the statistics can start to converge on the actual profile of the simulation. It can be difficult to get a good estimation of the time spent in processes whose total execution time is very short, which is a general limitation of statistical sampling, as explained in [11].

We compared the results of both approaches on our test cases. We ran the exact same simulation multiple times, with both profiling methods enabled on each of the 10 runs, so that the comparison is more reliable (separating the two types of profiling in different runs would add the unavoidable variability between runs), and we computed the dispersion of the results.

Table 1 shows in details the profiling data collected on 10 runs using the two profiling methods on the top contributor process in the memory controller design (TOP.mem_controller_0.smart_mem_q.send_transaction):

- Sample-based profiling, built-in in Incisive, reports a number of "hits" of the process. The equivalent time estimation is computed by dividing the number of hits by the frequency of the samples (333 hits/s).
- Time measured directly by the process activation and termination callbacks, accumulated from all the activations of the process.

We can notice that the estimated average time is lower for the callbacks-based method: this is a systematic bias, related to the profiling itself. The callbacks measure the time of the process execution only, while the sample-based profiling currently considers the callbacks as part of the process execution.

The dispersion is estimated by the coefficient of variation: the ratio of the standard deviation to the average. It is interesting to observe that the dispersion is 3.4x lower for the callbacks-based results, which tends to confirm that the direct time measurement from the callbacks is more accurate in this case.

TABLE I
MAIN CONTRIBUTOR PROCESS OF MEMORY CONTROLLER SIMULATION, ON 10 RUNS

	Sample-based		Time from callbacks
	# hits	equiv. time	
Sim run 1	2619	7.865	7.681326
Sim run 2	2571	7.721	7.601298
Sim run 3	2685	8.063	7.704177
Sim run 4	2589	7.775	7.614316
Sim run 5	2619	7.865	7.666795
Sim run 6	2620	7.868	7.701617
Sim run 7	2595	7.793	7.670353
Sim run 8	2678	8.042	7.668937
Sim run 9	2702	8.114	7.69941
Sim run 10	2573	7.727	7.599858
Average	**2625.1**	**7.883**	**7.661**
Std. dev.	**47.435**	**0.142**	**0.041**
Dispersion	**1.81 %**	**1.81 %**	**0.53 %**

The same analysis was made on the top 8 processes of the 10 runs of the memory controller design. Table 2 shows a summary of the average values found by each method, as well as the standard deviation and dispersion.

TABLE 2
TOP 8 PROCESSES OF MEMORY CONTROLLER SIMULATION, ON 10 RUNS

	Sample-based			Time measured by callbacks		
Process	average	std. dev.	dispersion	average	std. dev.	dispersion
mem_controller_0.smart_mem_q.send_transaction	7.883183	0.142448	**1.81 %**	7.6608087	0.0409176	**0.53 %**
im_2.generate_traffic	2.180781	0.036418	**1.67 %**	1.9783610	0.0251477	**1.27 %**
im_0.generate_traffic	1.967868	0.045213	**2.30 %**	1.8125638	0.0248937	**1.37 %**
im_3.generate_traffic	1.109910	0.071067	**6.40 %**	1.0332264	0.0122282	**1.18 %**
mem_controller_0.r_arb.transmit_responses	0.723123	0.048850	**6.76 %**	0.5104633	0.0082844	**1.62 %**
mem_controller_0.smart_mem_q.accept_requests	0.803303	0.050115	**6.24 %**	0.4081642	0.0070231	**1.72 %**
im_1.generate_traffic	0.406607	0.018195	**4.47 %**	0.3849621	0.0043943	**1.14 %**
mem_controller_0.r_arb.arbitrate_and_send_trans	0.769069	0.035148	**4.57 %**	0.3731437	0.0071083	**1.90 %**

The table shows that the observations on the top process are confirmed for the other processes of the simulation:

- Average time measured by the callbacks is lower, because it does not account for the profiling time itself, whose overhead increases with the number of activations (like in the case of the 8th process). The sample-based method can find closer values in case the callback-based profiling is not used, which confirms that results from both methods are reliable.
- The dispersion of the results is significantly lower in the results from the callbacks, especially for the processes with lower contribution, which are more difficult to evaluate accurately with sample-based profiling.

Overall, the lower dispersion of the callback-based profiling method can be very useful. Users are first interested in a view of the top processes that use a lot of their simulation time, but a rough estimate of the contribution is usually sufficient. However, when they start to optimize a simulation, or modify it for any reason, it is important to be able to compare one simulation run to another. Then the lower dispersion is precious, because it allows to measure modest incremental improvements, which are not lost in the noise of the measurement uncertainty.

V. CONCLUSION AND FUTURE WORK

In this paper we have proposed a complete interface that allows user's application code to be notified of all the state changes that occur in a SystemC process's lifetime, and execute a user-defined callback action, without a need for the user to modify the SystemC kernel. This interface is very generic and enables diverse and useful application areas, which we have demonstrated on a profiling case study.

In the simulation performance analysis domain, profiling solutions based on this process interface can produce very accurate results, which are useful for incremental optimization of model performance.

The natural next step is to propose the specification of this interface to Accellera System Initiative for future language standardization. Standardization of the process callback interfaces would allow to make models and applications relying on the callbacks compatible with other simulators.

References

[1] 1666-2011, IEEE SystemC Language Reference Manual, available at www.systemc.org, 2011.
[2] 1800, IEEE SystemVerilog Language Reference Manual, 2005.
[3] 1076, IEEE VHDL Language Reference Manual, 2008.
[4] ISO/IEC 14882:2011 "Programming Language. C++" - http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372
[5] SystemC Process and Event Tracing Interface proposal by Mentor Graphics Corp - http://www.eda.org/systemc/systemc-p1666/systemc-p1666-technical/hm/att-0159/ProcessTracing.h
[6] SystemC Process and Event Tracing Interface proposal reviews - http://www.eda.org/systemc/systemc-p1666/systemc-p1666-technical/hm/0160.html
[7] D. Becker, M. Moy, and J. Cornet, "Challenges for the parallelization of loosely timed SystemC programs", IEEE International Symposium on Rapid System Prototyping, 2015
[8] Cadence Virtual System Platform for the Xilinx Zynq-7000 All Programmable SoC - http://www.cadence.com/products/sd/virtual_system_Xilinx_Zynq/
[9] The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc - http://www.zynqbook.com/
[10] SystemC Verification Working Group (VWG) - http://www.accellera.org/activities/working-groups/systemc-verification
[11] J. Fenlason and R. Stallman, "GNU gprof, the GNU Profiler", Chapter 6 "Inaccuracy of gprof Output", 1993 - ftp://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_node/gprof_20.html