A Scriptable, Standards-Compliant Reporting and Logging Extension for SystemC

Jan Wagner, Rolf Meyer, Rainer Buchty and Mladen Berekovic
TU Braunschweig, E.I.S., D-38106 Braunschweig, Germany
Email: {wagner, meyer, buchty, berekovic}@c3e.cs.tu-bs.de

***Abstract*—The shift towards more and more complex System-on-Chips fosters high-level modeling (HLM) of new systems in order to provide required time-to-first-virtual-prototype and adequate simulation speed. Using HLM furthermore allows running exhaustive simulations are, enabling the developer to gain a plethora of information from the system during simulation. Reporting, logging, analyzing, and interpreting this vast amount of data requires a potent report and logging system. This paper proposes such a solution: the presented system is build on the foundations of SystemC's `sc_report` class and maintains full compatibility with it. To provide extensive search and analysis features, the proposed solution features Python-based scripting capabilities and supports attached key-value pairs to each report message. Using highly efficient black- and whitelisting filters empowers the user to reported events during runtime and suppresses all irrelevant reports in order to achieve fast simulation. Filter rules are fully scriptable and interpreted during simulation runtime, allowing dynamic adaption of the rules based on events occurred. All proposed mechanisms were evaluated under real-world conditions in an existing virtual prototype platform, including a report database backend, enabling easy analysis of the generated data.**

Keywords*—*SystemC, TLM, sc_report, report, logging, Python, scripting

I. Introduction

Modern embedded systems integrate large numbers of components on single chips. This development demands for new solutions to support efficient design of such Systems on Chip (SoCs), which may contain multiple processors combined with dedicated hardware. This trend is driven by the constant technological progress, which since the early 1970s has led to chips with ever larger capacities at lower costs. The strive for efficient utilization of the available silicon has triggered several paradigm shifts in system design: while in the early 1990s, VHDL and Verilog took over from schematic design, today, virtual platforms combined with techniques such as SystemC and Transaction-Level Modeling (TLM2.0) are about to further raise the abstraction level [1][2].

With raising the abstraction-level, new possibilities for gathering information from the simulated system were introduced. Not only is the simulation speed of a full-system simulation is increased significantly, but, in addition, high-level programming also offers plenty of options how to process and store information.

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no 621447 (Project "R5-COP").

SystemC and TLM speed up the simulation by several orders of magnitude relative to RTL simulations [3]. That high simulation speed allows running long simulations within reasonable time. Where simulation of a System-on-Chip booting an operation system based on an RTL-model can easily take several days, it can be performed on a SystemC/TLM-model in less than an hour. Such a simulation easily results in an enormous amount of events which can be stored in a file or database for subsequent analysis: given a RISC processor running at a simulated speed of 100 MHz, executing one instruction per clock cycle, booting an operating system in 30s results in an instruction log with 3 billion ($3 * 10^9$) logged events. Since not only one event per instruction is recorded, but other events as well, the recording mechanism's performance is critical. Furthermore, the recorded amount of data can exceed the size of several gigabytes very quickly. Hence, the recording framework must be aware of the performance impact on the simulation and able to reduce the amount of generated data with smart filters before it is processed and written to disk.

A logging mechanism suitable for SystemC/TLM simulations needs to consider these issues in an appropriate way: firstly, the impact on simulation performance must be as little as possible; secondly, the collected information must be prepared in order to aid the designer finding desired data during the analysis phase. Fostering the use of search queries on the logged data requires preserving the data type of additional information together with the main message text (e.g., an address) and its name. Another main concern is to pay attention to the source of the logging message. The location of the logging request in the source code is less important in this context. More important is the location within the *sc_module* hierarchy, since the very same code can be instantiated at several places in a SystemC/TLM simulation.

To give a sharp definition which particular issue the proposed solution addresses, the terminology of the terms *tracing*, *logging* and *reporting* needs to be clarified: *tracing* is defined in the SystemC standard [4, Chapter 8.1] as follows: "a trace file records a time-ordered sequence of value changes during simulation". The definition of *reporting* is given in the description of the *sc _report_handler*: "[...] *sc_report_handler* provides features for writing out textual reports on the occurrence of exceptional circumstances [...]". Finally, *Logging* is a feature of reporting and is described as writing report messages into a file. Following these definitions, the proposed extension addresses reporting and logging in SystemC simulations, tracing is not considered in this work.

The remainder of this paper is structured as follows: Section II presents an overview on related work; Section

III explains the proposed logging extension. Section IV is dedicated to the technical details, Section V contains the evaluation, and in Section VI, conclusions are drawn.

II. Related Work

In the past years, we have used a straightforward logging system (called verbosity kit) within our SystemC/TLM2.0 virtual platform [5]. It is based on regular C++ streaming to *stdout*, extended by some SystemC-related features such as adding the current simulator time and delta cycles to each line, support of five distinct message types (debug, info, report, warning, error), a predefined place for the module name, and text coloring for readability purposes. Furthermore, performance issues were addressed by making use of compiler optimizations. The log level is defined as a constant value enabling the compiler to remove never-executed logging code at compilation time. However, the log level is required to be defined before compilation, hence the whole system needs to be recompiled after it was changed. While performing a validation of our virtual platform against the RTL simulation of the corresponding hardware, we encountered the limits of this logging approach: we had to compare the time of occurrence of several 100.000 events from both simulations, including additional information like addresses. It turned out that parsing several gigabytes of log files with Python scripts can be very time-consuming, since parsing of data is very slow. From this experience we formulated the following requirements for a logging framework:

(a) Usability improvement:
- Fine-grained control at runtime of which logging statements are required in order to reduce the amount of recorded data without recompilation
- Clear coding style, since we realized most logging statements using stream operators are hard to read. Further discussions of this can be found in [6].

(b) Analysis support:
- Possibility to attach additional information to certain events as a key/value pair preserving the data type to aid subsequent analysis.
- Flexible storage backend in order to support subsequent analysis.

(c) Overhead reduction: reduce the overall logging overhead

(d) Compatibility: take into account the SystemC/TLM standard [4] in order to maintain compatibility to TLM components of other vendors

Considering these requirements the following solutions were evaluated. A general overview of text-based logging including theoretical foundations is given in [7].

Most logging frameworks are bound to a certain use-cases, for example the presented solutions by Kraft et.al. [8], Hung et. al. [9] or demonstrated with TinyTLS [10] are designed for embedded systems and their very limited resources. Since in SystemC/TLM simulations the resources are not the bottleneck, but performance and data processing, these frameworks do not satisfy our needs. Other frameworks are tailored to specific system architectures. Verbowski et. al. [11] describes an approach for logging events on monitored clients and analyzing them on a central observer. Anderson et. al. [12] proposes a solution for a similar scenario. Other techniques require a lot of manual work to integrate them: Aleekseev et. al. [13] utilizes the control-flow graph for finding a set of monitoring points, which reduce the impact to the program execution to a minimum. The result of the process would support the requirements of a SystemC/TLM simulation, but applying this approach is not feasible.

A couple of other general purpose C/C++ logging libraries exist, which could be used in conjunction with SystemC/TLM: VampirTrace's focus is in the high-performance computing domain [14], especially parallel programs. This includes tracing function enter and leave events, MPI communication, OpenMP events, and performance counters. Technically, this is realized by code instrumentation. Unfortunately it is only capable to perform tracing, it does not respect the inherent structure of a SystemC simulator, and, moreover, SystemC simulations neither use OpenMP nor MPI communication at all.

log4cxx [15] is a C++ clone of the de-facto Java standard logging utility *log4j*. At the first glance, it is a good candidate to be used in SystemC simulators. It supports hierarchical loggers, which surely can be mapped to the *sc_module* hierarchy in SystemC simulators. Moreover, it has built-in support for message filtering regarding their importance, like DEBUG, INFO, or WARNING level. However, it has shortcomings regarding the other postulated requirements: there is no fine-grained control on logging during runtime, and no support for attaching key/value pairs to reports. This leads to the same unhandy coding style with streaming operators as with the verbosity kit. Furthermore, it does not integrate with `sc_report` as described in the SystemC/TLM standard [4]. CULT [16] claims to be a tracing and logging framework for SystemC, it focuses on tracing using the definition given in IEEE-1666 [4]. The reporting features of CULT have similar drawbacks as *log4cxx*: no fine-grained control during runtime, no possibility to attach additional information, resulting in an unclear coding style by attaching information manually. Despite it is explicitly designed for SystemC, it is not compatible to *sc_report*, which is the standard reporting mechanism defined for SystemC [4, Chapter 8.2]. *sc_report* comes with a lot of features, but is neglected by the SystemC community since it has some drawbacks: a minor one is that the content of the message-type field needs to be filled with adequate values by the programmer at every logging request in the source code. The standard is furthermore vague regarding which information to put in there. More important to us is that no data can be appended to a log message, e.g., the address of a request to the system bus, except converting it to a string. Another major drawback from analysis point of view is the lack of appropriate mechanisms for runtime control over when messages should be generated and when not in order to reduce the overall amount of messages. Even with having such a mechanism, huge amounts of messages are generated; hence, a more flexible storage backend implementation is required to store data in a well-structured manner.

The approach presented in this paper is based on the foundations of *sc_report* and will overcome the impairments of *sc_report*. It extents the functionality of the *sc_report* feature, while maintaining full compatibility to the standard [4].

III. Background and proposed API

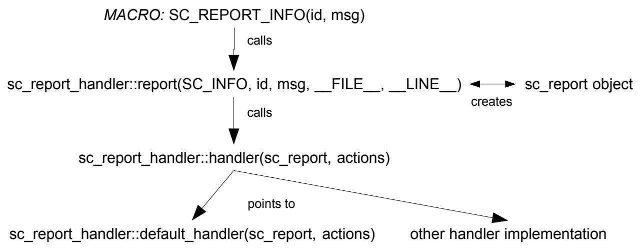
MACRO: SC_REPORT_INFO(id, msg)
calls
sc_report_handler::report(SC_INFO, id, msg, __FILE__, __LINE__)
creates
sc_report object
calls
sc_report_handler::handler(sc_report, actions)
points to
sc_report_handler::default_handler(sc_report, actions)
other handler implementation

Fig. 1: Actions triggered by a report request

A. *Background of sc_report*

`sc_report` is an object representing a report. The reporting integrated in SystemC offers macros for creation of these objects. For predefined severity levels, macros are defined: `SC_REPORT_{INFO, WARNING, ERROR, FATAL}(id, msg)` with `id` as a so-called message type, which is a string and the message itself. Additionally, `SC_REPORT_INFO_VERB(id, msg, verbosity)` allows manual setting the verbosity between 0 and 500. [4] recommends to build the `id` following this pattern: *"/originating_company_or_institution/product_identifier/-subcategory/subcategory..."*. This recommendation unfortunately misses the aspect that a very same component can be instantiated multiple times at different places within a SystemC simulator, and the suggested information therefore cannot be derived from information, which is available anyway in the platform. Hence, we automatically fill in the `sc_module` name, including the full hierarchical path, to create a unique id within the system pointing to the report-issuing component.

Beyond this, an *sc_report* contains the filename and line where it was created in the source-code, the *sc_process*-name and of course the *sc_time_stamp* when the event occurred.

B. *Background of sc_report_handler*

sc_report_handler provides features for handling *sc_report* objects. Not only methods for generating new reports are provided, but also mechanisms for managing generated reports. The report handler holds a verbosity threshold-level: reports with a higher verbosity-level than the threshold are ignored. Another important mechanism is that one or more actions can be assigned to every report message. Actions define what should happen with reports matching certain criteria. Criteria are severity level, message type, or a combination of both. For each severity level, default actions are defined which the programmer can alter. Valid actions are `SC_UNSPECIFIED, SC_DO_NOTHING, SC_THROW, SC_LOG, SC_DISPLAY, SC_CACHE_REPORT, SC_INTERRUPT, SC_STOP, SC_ABORT`. SystemC provides a replaceable default handler for executing the actions assigned to every report message.

C. *Building blocks of proposed SystemC logger*

The proposed solution is composed of four components addressing different aspects of the requirements. Each component will be discussed separately.

Frontend/SystemC API/Extension of sc_report: This paragraph describes the interface for generating reports. With listing 1, an example is provided how the programmer can use the proposed framework.

Listing 1: SystemC example

```
srInfo(<optional id>)
  ("key1", value1)
  ("key2", value2)
  ("message");
```

In line one, the report's severity level is defined by choosing one of the predefined levels or user-defined types. Predefined types are `srInfo`, `srWarn`, `srError`, and `srFatal` representing the standard severity levels of *sc_report*. Additionally, we defined `srDebug`, `srAnalyse` and `srMessage`: these are of `srInfo` severity level, but offer different levels of verbosity. Each function takes an id string as its optional argument, except `srMessage`, which in addition takes a non-optional parameter manually defining the verbosity level (like `SC_REPORT_INFO_VERB`). If no id is given, the hierarchical *sc_module_name* is inserted automatically.

Line two and three define optional key/value pairs attached to the report message. There is no limit on how many pairs can be attached. The last parentheses in line four contains the mandatory message text.

Extension of sc_report_handler: sc_report_handler was extended with respect to the additional key/value pair, which now can be attached to the reports. Beyond that, the filtering mechanism was improved regarding flexibility and speed. Filtering now supports black- and whitelisting of *sc_objects*. For `id` search, a speed-up was achieved by using a fast pointer comparison instead of a string comparison like in the default handler.

A new report handler function for sc_report_handler: The default handler shipped with SystemC was replaced by an entirely new component. It takes all incoming report messages, including all information stored in the *sc_report*-object, and passes it into a Python runtime environment where the actual handler is implemented.

Python backend: The Python backend is very straightforward. A function can be registered within a regular Python script to be called when a report needs to be processed. The function needs to take certain arguments as shown in listing 2.

Listing 2: Python backend example

```
def report(
    message_type=None,
    message_text=None,
    severity=None,
    file_name=None,
    line_number=None,
    time=None,
    delta_count=None,
    process_name=None,
    verbosity=None,
    what=None,
    actions=None,
    phase=None,
    **kwargs):
```

The filtering mechanisms in the *sc_report_handler* are also accessible from the Python script. The verbosity level can be set with `set_verbosity(int level)`; the white- and blacklisting filters can be configured with `set_filter_to_whitelist(bool value)`, `add_sc_object_to_filter(name, severity, verbosity)`, and `remove_sc_object_-from_filter(name)`, where name is the full *sc_module*-name including the path in the module hierarchy.

IV. Technical details

A. API implementation

The `sc_report` object contains a field for the file name and line number of the report issuing location in the source code. This information is not directly available to C++ code as a variable, but the preprocessor provides this information during macro interpretation as values through constants `__FILE__` and `__LINE__`. Hence, macros are essential to implement the API.

To ensure good usability, the API user should be freed from typing the same code over and over again, but influence the outcome when necessary. Therefore, the input of `id` should be optional. If no input is given, the hierarchical `sc_object` name is filled in. Unfortunately, no macro overloading is available, but its behavior can be emulated with clever combination of available features as shown in Listing 3.

Listing 3: Macro implementation

```
19 #define _GET_MACRO_(dummy,_1,NAME,...) NAME
20
21 #define srInfo(...) _GET_MACRO_(\
22   dummy,\
23   ##__VA_ARGS__,\
24   srInfo_1(__VA_ARGS__),\
25   srInfo_0())
26
27 #define srInfo_0()\
28     sr_report_handler::report(\
29       sc_core::SC_INFO,\
30       this,\
31       this->name(),\
32       "",\
33       sc_core::SC_LOW,\
34       __FILE__, __LINE__)
35
36 #define srInfo_1(id)\
37     sr_report_handler::report(\
38       sc_core::SC_INFO,\
39       NULL,\
40       id,\
41       "",\
42       sc_core::SC_LOW,\
43       __FILE__, __LINE__)
```

If `srInfo()` appears in the source code, the following actions happen during macro evaluation: the helper macro `_GET_MACRO_` is used to decide whether the `srInfo_0` or the `srInfo_1` should be chosen. This happens by putting the arguments `__VA_ARGS__` before the macro names to be executed (line 23 to 25). The double hash `##` suppresses the trailing comma. Hence, the to be executed macro shifts to the third position in the arguments list of `_GET_MACRO_`. In case no argument is passed to `srInfo`, line 25 represents the third argument; in case one argument is passed, line 24 represents the third argument.

The bracket-chaining interface for adding an arbitrary number of key/value pairs is built upon the techniques shown in [17] for the `desc.add_options` method. Technically, it is based on overloading of the *()* operator, where each bracket with a key/value pair returns a pointer of the current instance of the `sr_report` class. This way, an unlimited number of key/value pairs can be attached, until the last bracket only contains the message string.

B. SystemC and Python integration

The Python-to-SystemC interface was implemented using the standard Python/C API as described in [18]. Basically, the SystemC program calls a Python function registered as a callback function to the simulation's C++ part. To this function, all standard fields of `sc_report` as defined by IEEE-1666 [4] are transferred as parameters, whereas the optional key/value pairs are passed in a dictionary to preserve the relation between the key name and its value.

V. Evaluation

A. Performance test

The test setup for the reporting system was based on a complete System-of-Chip simulation: the centerpiece of this SoC is a Leon 3 processor, which was created by Aeroflex/Gaisler as a Sparc V8 derivative for the European Space Agency [19]. To constitute a complete SoC, accompanying peripherals are required. Most important components are the AMBA bus module connecting the Leon core with its MMU and memory controller via AHB, and further peripherals like interrupt controller (IRQMP), timer module (GPTIMER), and the UART device via APB. All mentioned components are available as VHDL models from Gaisler and also as SystemC/TLM2.0 models as described by Schuster et. al. [5], available on GitHub [20]. Since reporting does only make sense when the simulator is fed with software, a synthetically benchmark was run to trigger the reports. According to the simulator, this benchmark would require 48.2 s to finish on real hardware with a 100 MHz clock. To stress the reporting part of the simulation, each and every memory access was reported in the memory management unit (MMU) and in the memory model, including with address and data request length. This results in more than 211 million report messages during simulation execution, causing log files of 24 GB or a database of 8.8 GB size. This configuration can be considered as worst case, because usually less frequent events are reported or the reported events are limited to certain components like the MMU. All simulations were performed on an Intel i7-4790 CPU equipped with 16 GB RAM and an SSD drive.

Firstly, the effect of introducing the proposed framework into the simulation was determined. Therefore, a simulation was run without reports, and the very same simulation was run reporting all memory accesses on MMU and memory level, while the black- and whitelisting filters suppressed all reports. As depicted in Figure 2, the overhead introduced by this scenario is negligible. In our tests, the simulation takes only approximately 10% longer with suppressed reports than

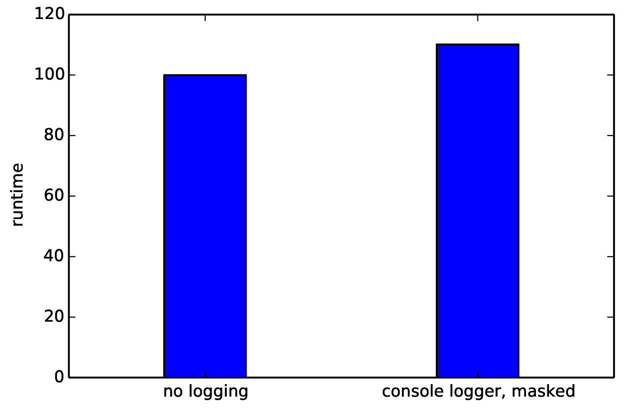

Fig. 2: Performance impact while suppressing all reports

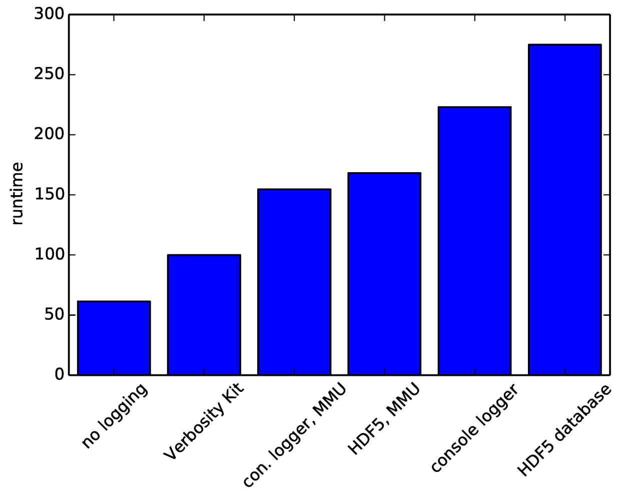

Fig. 3: Overhead introduces with different configurations

without any reporting. This is important, because reporting statements can reside in the code and can be activated on demand by a script without significant impact on the overall run-time.

The next setup compares different configurations, which occur in everyday work with this framework. The described scenario is a real example and describes how the proposed solution was used for bug hunting in our virtual platform. We assume a bug in the MMU and we want to figure out what is going wrong. With the old verbosity kit, either the source code needs to be altered or all reports are logged. Since we do not want to disable all reports spread over the whole project in the sources, every debug report is logged. In our setup, reports from MMU and memory are logged. The second bar in Figure 3 (*Verbosity Kit*) represents the time required to do so. For comparison the first bar (*no logging*) represents the time required to execute the simulation without any logging.

Writing all reports into a text file, like verbosity kit, using the new approach is depicted with the fifth bar (*console logger*). The sixth bar (*HDF5 database*) represents the time it takes to store everything into a database. Compared the Verbosity-Kit-

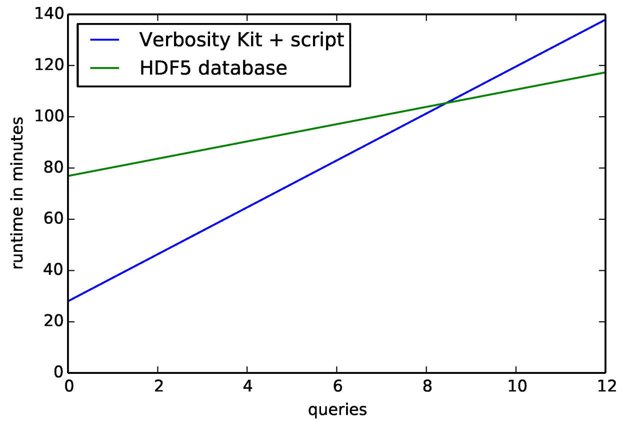

Fig. 4: Comparison of complete typical use case

approach the simulation runtime is prolonged by a factor of 2.2 and 2.7, respectively. But since the faulty behavior is expected in the MMU, only the reports issued from the MMU need to be stored. At this point, the black- and whitelisting filters come into play: putting only the MMU into the whitelisting filter, the factor can be reduced to 1.5 (third bar, *con. logger, MMU*) for logging into a text file and to 1.7 (fourth bar, *HDF5, MMU*) for database storage.

Let's assume the MMU seems to work well, but in some cases something goes wrong. Hence, particular reports need investigation and therefore must be filtered out from several million reports. We want to have a look at all reports regarding a certain address range. The process of extracting this information is quite different on using the text log file from verbosity kit and the database generated by the proposed approach. A customized script was created to parse the text file and storing all matching reports into another file. In our test, one run through the 24 GB log file takes more than 9 minutes. In contrast to that, querying the database is simple and fast: only the query expression needs to be written down and sent to the database interface. Getting the same data as from the text log, takes only a bit more than three minutes. As depicted in Figure 4 one will save time using the database approach, if more then eight queries are sent to the same dataset, despite creating the database takes more time than creating a text log file.

B. Results

Evaluation data proves that the proposed framework introduces only negligible overhead compared to no logging, as long as the filters are activated and no report processing is triggered. This is in contrast to the significant overhead introduced by existing solutions performing report processing: the `stdout`-based verbosity kit was designed for high performance and comprises therefore no components, causing high processor load. This was done at the cost of having no configuration options after the compilation process. One of the main features of the proposed solution is configurability even at runtime. Hence, an advanced message handling is required during runtime, causing a slowdown of the simulation time by a worst-case factor of 2.2, i.e., if all reports are printed as with verbosity kit. Using the runtime-configurable white- and blacklisting filters, this impact can be considerably reduced to the absolute necessity by activating verbose reporting if required and deactivating it afterwards. The process of activation and deactivation can be triggered by report messages or by static rules, such as after reaching a predefined time in simulation verbose logging is activated.

Moreover, messages can easily be processed with all options Python offers. For our work, it was very beneficial to store all reports into a database, allowing queries on the produced data fostering easy analysis. As database, HDF5 [21] was chosen because of its capability to handle large amounts of data and good Python integration. The database itself was designed for databases with enormous growth in size and complexity. The advantage of having fast and easy access to the data comes at a worst-case 2.7-times slowdown compared to plain `stdout` logging if all messages are stored. As with the console logging, the slowdown can be mitigated using the white- and blacklisting filters.

All benchmarks presented in this paper were performed with a single-threaded simulator, which results in sequential processing of the simulation and the reporting and logging components. Using recent multi-core hardware, we expect the overhead can be reduced significantly when report handling, like preprocessing, printing or storing into the database, are shifted into separate threads. Multi-thread support is currently worked on.

C. Requirements evaluation

All requirements stated at the beginning of this paper were satisfied.

(a) Fine grained control at runtime is available through the Python script interface in conjunction with black- and whitelisting filters.
(b) Key/value pairs can be attached to every report.
(c) As storage backend, everything Python interfaces to is possible. Console reporting and database storage were implemented.
(d) The overhead was kept as low as possible through efficient implementation of black- and whitelisting.
(e) Clear coding style is supported by bracket-chaining API and automatic insertion of the `id` if desired.
(f) Full compatibility to the SystemC standard is maintained. All reports using the standard functionality of `sc_report` are treated in the same way as the extended reports.

VI. Conclusion

A standard-compliant extension of `sc_report` was proposed, compared to the standard approach and benchmarked with an analysis of reports for a real debugging case. The default reporting tools of SystemC were extended with a comfortable scripting interface and support for smart handling of key/value pairs attached to reports. The scripting interface relies on the Python scripting language and the widespread libraries available for it. With the introduction of efficient black- and whitelisting mechanisms, the simulation time impact can be reduced to a minimum, while still benefitting from the capable scripting interface. Further speedup is expected with introduction of multi-thread support, which will be available in the near future.

References

[1] Kun Lu, Daniel Müller-Gritschneder, Ulf Schlichtmann, "Accurately Timed Transaction Level Models for Virtual Prototyping at High Abstraction Level," in *Design Automation and Test in Europe (DATE)*, 2012.

[2] A. Gerstlauer, S. Chakravarty, M. Kathuria, P. Razaghi, "Abstract system-level models for early performance and power exploration," in *The 17th Asia and South Pacific Design Automation Conference*, 2012, pp. 213–218.

[3] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems.* Springer, 2005. [Online]. Available: http://researchbooks.org/0387262326

[4] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[5] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "SoCRocket – A virtual platform for the European Space Agency's SoC development," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, May 2014, pp. 1–7.

[6] Google Style guide for C++. [Online]. Available: https://google-styleguide.googlecode.com/svn/trunk/cppguide.html#Streams

[7] J. Valdman, "Log File Analysis," *Department of Computer Science and Engineering (FAV UWB)., Tech. Rep. DCSE/TR-2001-04*, 2001. [Online]. Available: https://www.kiv.zcu.cz/site/documents/verejne/vyzkum/publikace/-technicke-zpravy/2001/tr-2001-04.pdf

[8] J. Kraft, A. Wall, and H. Kienle, "Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects," in *Runtime Verification*, ser. Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace *et al.*, Eds. Springer Berlin Heidelberg, 2010, vol. 6418, pp. 315–329.

[9] S.-H. Hung, S.-J. Huang, and C.-H. Tu, "New Tracing and Performance Analysis Techniques for Embedded Applications," in *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on.* IEEE, pp. 143–152.

[10] R. Sauter, O. Saukh, O. Frietsch, and P. J. Marrón, "TinyLTS: Efficient Network-Wide Logging and Tracing System for TinyOS," in *INFOCOM, 2011 Proceedings IEEE.* IEEE, 2011, pp. 2033–2041.

[11] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee *et al.*, "Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 117–130.

[12] E. Anderson, C. Hoover, X. Li, and J. Tucek, "Efficient tracing and performance analysis for large distributed systems," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on.* IEEE, 2009, pp. 1–10.

[13] S. Alekseev, "Algorithms for Optimally Tracing Time Critical Programs," in *The 2006 International Conference on Software Engineering Research and Practice.* Citeseer, pp. 26–29.

[14] M. Jurenz. TUD-ZIH – VampirTrace. [Online]. Available: http://www.tu-dresden.de/zih/vampirtrace

[15] Short introduction to Apache log4cxx. [Online]. Available: http://logging.apache.org/log4cxx/

[16] W. Hong, A. Viehl, N. Bannow, C. Kerstan, H. Post, O. Bringmann *et al.*, "CULT: A unified framework for tracing and logging C-based designs," in *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, Sept 2012, pp. 1–6.

[17] Bracket chaining tutorial. [Online]. Available: http://www.boost.org/doc/libs/1_55_0/doc/html/program_options/-tutorial.html

[18] Python/C API Reference Manual. [Online]. Available: https://docs.python.org/2.7/c-api/index.html

[19] Aeroflex/Gaisler IP and manual download. [Online]. Available: http://www.gaisler.com/index.php/downloads

[20] SoCRocket sources. [Online]. Available: https://github.com/socrocket

[21] The HDF group. [Online]. Available: http://www.hdfgroup.org