

A Fast SystemC Engine

Daniel Gracia Pérez Gilles Mouchard[†] Olivier Temam

ALCHEMY INRIA Futurs & LRI, Paris South University

[†] CEA LIST

France

Abstract

SystemC is rapidly gaining wide acceptance as a simulation framework for SoC and embedded processors. While its main assets are modularity and the very fact it is becoming a de facto standard, the evolution of the SystemC framework (from version 0.9 to version 2.0.1) suggests the environment is particularly geared toward increasing the framework functionalities rather than improving simulation speed. For cycle-level simulation, speed is a critical factor as simulation can be extremely slow, affecting the extent of design space exploration.

In this article, we present a fast SystemC engine that, in our experience, can speed up simulations by a factor of 1.93 to 3.56 over SystemC 2.0.1. This SystemC engine is designed for cycle-level simulators and for the moment, it only supports the subset of the SystemC syntax (signals, methods) that is most often used for such simulators. We achieved greater speed (1) by completely rewriting the SystemC engine and improving the implementation software engineering, and (2) by proposing a new scheduling technique, intermediate between SystemC dynamic scheduling technique and existing static scheduling schemes. Unlike SystemC dynamic scheduling, our technique removes many if not all useless process wake-ups, while using a simpler scheduling algorithm than in existing static scheduling techniques.

1. Introduction

SystemC [5] is rapidly gaining wide acceptance as a simulation framework for SoC and embedded processors. Its main assets are the modular structure of the simulators which facilitates component reuse and sharing, the ability to combine cycle-level modeling and transaction-level modeling, and now the very fact that it is being used by an increasingly large community. In this article, we are particularly interested in cycle-level modeling, and especially

the performance issues of cycle-level modeling using SystemC; as we all know, cycle-level simulation can be extremely slow so that any means for speeding up simulation can have a direct and strong impact on design-space exploration. While the SoC community appreciates the combination of transaction-level and cycle-level modeling, both the SoC and processor architecture communities have a need for extensive cycle-level modeling. While SystemC naturally supports cycle-level modeling, the evolution of the SystemC framework (from version 0.9 to version 2.0.1) suggests the environment is particularly geared toward increasing the framework functionalities for transaction-level modeling, especially the different communication means between modules, rather than improving simulation speed. So architecture designers that need fast simulation frameworks can either turn to other environments or improve the speed of SystemC. Now that the embedded systems community is finally seeing a simulation standard emerging, albeit a *de facto* standard like SystemC, it would be a step backward to investigate or attempt to promote another standard for fast cycle-level simulation, so our research group has been investigating methods for speeding up SystemC simulation, especially cycle-level simulation. For that purpose, we have written a new SystemC engine and achieved speedups ranging from 1.93 to 3.56 over SystemC 2.0.1 on two processor simulators, by improving the engine implementation and the scheduling algorithm. These experiences were run on a workstation with a Pentium 4 2Ghz. For the moment, our SystemC engine only supports the subset of the SystemC syntax most often used for cycle-level simulators, but we intend to support other syntax constructs in the future if needs be.

We decided to write a new engine from scratch instead of stripping the existing SystemC engine of unsupported features and modifying the engine behavior because the current engine implementation has obviously not been optimized for performance. We applied a set of software engineering techniques to clean up the engine implementation and obtained a faster SystemC engine with speedups ranging from

1.60 to 2.44 over SystemC 2.0.1.

After eliminating implementation inefficiencies from the SystemC engine, we focused on the more complex task of improving the scheduling algorithm. While a poor SystemC engine implementation has a strong but uniform impact on simulation speed, the SystemC scheduler has a much more irregular impact. The number of process wake-ups can have a strong impact on performance, and whether processes are waken up naturally depends on the signals to which a process is sensitive (the process is waken up if the signal value changes). With the SystemC dynamic scheduling technique the number of times a given process is waken up *within* a simulated clock cycle can strongly vary (from 1 up to 12 times within our experiments). Because the occurrence of such wake-up peaks is hard to predict by the simulator programmer and because they can strongly degrade simulation performance, SystemC is in fact fairly unreliable performance-wise. Consequently, we have implemented a new SystemC scheduler in order to smoothen and improve SystemC performance. Unlike static scheduling techniques [4, 2, 6], our scheduling technique, called *acyclic scheduling*, does not require a complex (and sometimes lengthy) analysis of the process call graph, only simple and fast code generation and compilation of the scheduler before starting the simulation. We experimentally show that this scheduler avoids most multiple process wake-ups within a single simulated clock cycle, and we achieve an additional speedup of 1.21 to 1.46 over our improved SystemC engine for the simulators we tested.¹

In Section 2, we present our new SystemC engine implementation, how it differs from the original SystemC engine implementation and performance comparisons. In Section 3 we present our new scheduling technique, its implementation, and performance comparisons with our SystemC engine and the original SystemC engine.

2 A performance-oriented SystemC engine

The SystemC 1.0 scheduler is a loop that starts with a list of all processes sensitive to the clock. When a process is waken up, it may write to one or several output ports (`sc_out`) connected to signals (`sc_signal`), themselves connected again to input ports (`sc_in`) in other modules. When it writes to an output port, it adds the corresponding signal in a list of active signals (a LIFO queue). After all processes have been executed, a second iteration, called a *delta cycle*, starts again: the list of signals is scanned and all processes sensitive to the active signals are waken up and so on; the simulated clock cycle ends when the list of active signals is empty at the end of a delta cycle. The scheduler of

¹This new SystemC engine with improved implementation and scheduling is now freely distributed at <http://www.microlib.org/Core>

SystemC 2.0 is somewhat different because signals are implemented using channels, themselves implemented using events. When a process writes to an output port, it updates a channel which itself creates an event for the target process. The list of active signals is replaced with a list of events (a FIFO queue).

Subset of SystemC syntax. To implement fast cycle-level simulators, many researchers and engineers have turned to SystemC methods (`sc_method`) and signals. Methods have been preferred over threads (`sc_thread`) because they are much faster as pointed out by Charest et al. [1]. Signals have gained wide acceptance because they were the sole communication means of earlier SystemC versions and because they are a rather intuitive model of the hardware links between logic blocks, compared to *channels* and *events*. Consequently, the syntax constructs supported by our SystemC engine are `sc_method`, `sc_in`, `sc_out` and `sc_signal`. If needs be, we will implement events and other constructs in future versions of our engine. Naturally, all simulators developed using this subset of the SystemC syntax can be executed with the original SystemC 1.0 and 2.0 engines.

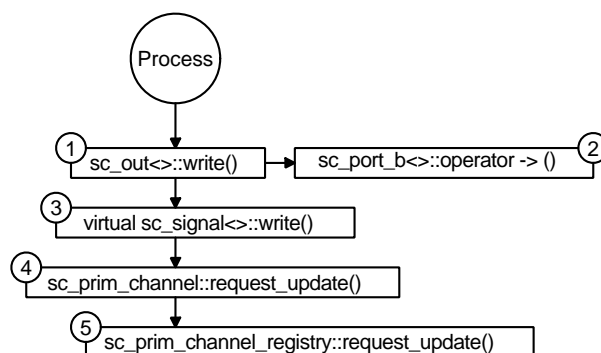


Figure 1. Nested method calls.

A tuned implementation of the SystemC engine. Fine-tuning the implementation of the SystemC engine loop may seem picky but it can have a great impact on performance, especially if the simulator is fairly modular and little time is spent in each process. So as to maximize performance and avoid all unnecessary code, we wrote the engine from scratch rather than improve current engines. We brought the following set of software engineering improvements over the original SystemC engine:

- The original SystemC engine makes an extensive use of nested virtual method calls. For instance, the construction of the active list of `channels` needs multiple nested method calls, see Figure 1: (1) `Process` calls the `write` method of the output port, (2) the port retrieves the primitive `channel`, i.e.

the signal connected on the port, (3) the write method of the signal (which is virtual) is called, (4) the signal requests an update calling a method of its base class `sc_prim_channel`, (5) the request update to the primitive channel registry ends the call chain. A virtual method call is costly because the code generated by the C++ compiler has to perform a table lookup for each call. Such calls are usually not inlined by the compiler. Consequently, we overrode the compiler inlining algorithm and manually inserted inlining pragmas wherever necessary.

- In the original SystemC engine implementation, a signal has two buffers associated with each port: a write buffer for the output port and a read buffer for the input port. When a signal is added to the list of active signals, the signal value is propagated from its input port to its output port. This propagation is implemented as a copy of the write buffer to the read buffer. We replaced this copy with a toggle to swap the role of the read and write buffers and avoid data movements.

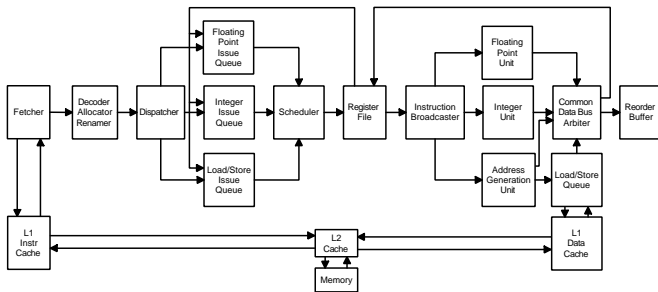


Figure 2. Simulated 4-way superscalar processor.

Experimental evaluation. We tested this modified implementation on two simulators. The first simulator models a simple RISC architecture much like the MIPS R2000 [3]. The second simulator corresponds to a full superscalar processor much like an HP Alpha 21264 with 9 pipe stages, 4-way, 12 functional units, 128 physical registers, and non-blocking caches, see Figure 2; it has 7 stages for integer instructions, 8 for load/store instructions, and 9 for floating point instructions; it also has 3 branch predictors (a gshare branch predictor, a branch target buffer and a return address stack), it performs out-of-order execution and store result forwarding. The modified engine implementation brings a speedup of 2.44 for the RISC simulator, and 1.92 for the superscalar simulator, see Figure 7. While the RISC processor has 29 processes and the superscalar processor has 54 processes, the architecture modeled by the superscalar simulator corresponds to almost 300 times more transistors than the architecture modeled by the RISC simulator; as a result, the superscalar processes are significantly bigger and more

complex than the RISC processes. More generally, the bigger the processes the smaller the influence of the SystemC engine on performance, or conversely, the more modular the simulator the bigger the influence of the SystemC engine on performance.

3 Acyclic Scheduling

SystemC simulators are typically made of sequential processes which are sensitive to a clock edge, and combinational processes which are sensitive to their input ports. The former processes are typically waken up at the beginning of the simulated clock cycle; the order in which the latter processes are waken up will determine the total number of wake-ups within the simulated clock cycle. This order is called the process *schedule*; finding the optimal process schedule is not a novel issue, and there is much work on this topic for other environments and languages like VHDL and more recently Liberty [8, 6], a modular framework for processor modeling. SystemC proposes a dynamic scheduling mechanism: at each simulated clock cycle, processes which are sensitive to the clock front edge are waken up, and during their execution, they modify outgoing signals, waking up in turn other processes. A simulated clock cycle is over when no more process needs to be waken up, possibly after multiple iterations/delta-cycles. This technique is called *dynamic scheduling* because the set and order of process to wake-up is not known at the beginning of the simulated clock cycle. While dynamic scheduling is fairly easy to implement, it also suffers from an excessive number of wake-ups. To better understand how process scheduling can affect the number of wake-ups, consider the example of Figure 3.

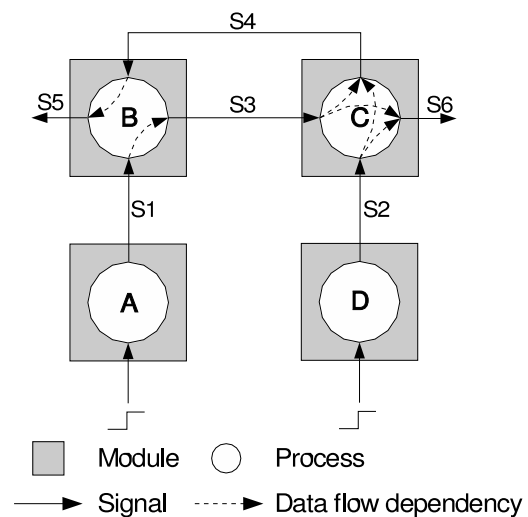


Figure 3. Example SystemC simulator with 4 processes.

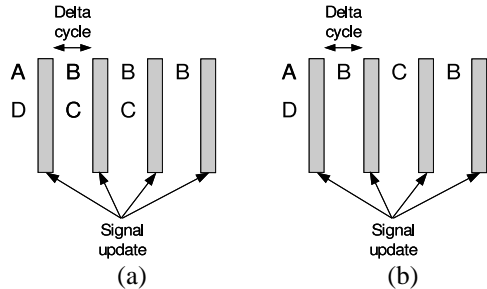


Figure 4. (a) *Suboptimal* and (b) *Optimal Schedules*.

The architecture is composed of 4 modules with one process each. Processes A and D are sensitive to the clock front edge; Process A updates signal S1 and Process D updates signal S2. Process B is sensitive to signals S1 and S4, and updates signals S3 and S5. Process C is sensitive to signals S2 and S3, and updates signals S4 and S6. The dotted arrows in Figure 3 indicate the data flow within each process: for instance, when S2 is modified, S4 and S6 are modified by Process C. Suppose that signal S3 is a request and signal S4 is an acknowledge: Process B must assert signal S3 before gaining an acknowledge on signal S4. Figure 4a shows how SystemC could schedule such a set of processes. (1) Processes A and D wake up because they are sensitive to the clock front edge. The scheduler updates signals S1 and S2. (2) Processes B and C wake up. The scheduler updates signals S3, S4, S5 and S6. (3) Processes C and B wake up. The scheduler updates signals S4, S5 and S6. (4) Process B wakes up. The scheduler updates signals S5. We can observe that Process B wakes up three times, and Process C wakes up twice. If Process C were waken up after Process B has executed, and then Process B were waken up again, the total number of process wake-ups would be 5 instead of 7, see Figure 4b. Therefore, if process wake-ups were scheduled in a given order and serialized, we could avoid many useless wake-ups.

Seeking an appropriate wake-up order for processes is the principle of *static scheduling*, an alternative to dynamic scheduling: the architecture is viewed as a graph, each process being a vertex and each link an oriented edge (or conversely), and the graph is analyzed to determine the shortest possible path through the graph; this path corresponds to a process schedule which is then *compiled*. The biggest difficulty with static scheduling is breaking *graph cycles*: in practice, cycles usually do not exist within a clock cycle, but looking at the architecture graph, process dependencies can give the impression that such cycles exist, so that it is not possible to determine where to start and end process wake-ups. Such false cycles can be eliminated if the programmer provides some information on the dependencies between signals. Consider again the example of

Figure 3 where there is a cycle between Processes B and C. Assume now the scheduler is aware of the data flow dependencies denoted by the dotted lines within B and C; then the cyclic graph becomes an acyclic graph and it appears signals should be computed in the following order: S1 and S2, then S3, then S4 and S6, then S5; as a result the proper process wake-up order is A and D, then B, then C, then B, i.e., the schedule of Figure 4b.

Several different methods for finding an appropriate schedule have been implemented in various simulation environments, and they almost all rely on complex graph analysis techniques; usually, processes are represented as vertices and signals as edges (or conversely), and the graph is oriented since signals usually have an input and an output port. Hommais et al. [4] propose to compute the graph strongly connected components (graph subsets where there is path from each vertex to any other vertex), and within each path from each vertex to any other vertex), and within each component, to generate a schedule for the vertices. For that purpose, they find an hamiltonian path (a path visiting each vertex exactly one time) or create an arbitrary path if it does not exist. Then the components themselves are sorted using Tarjan algorithm [7] and executed in that order. For each component, a relaxation algorithm is applied: looping on the hamiltonian path until it converges, i.e., until no new output is produced by any vertex of the component, then the next component is executed and so on. In other words, the scheduling is static within each component with a backup dynamic scheduling. Edwards [2] uses a similar method with some differences. The major difference is that it is assumed that processes compute their outputs at most once per cycle. This hypothesis makes it hard to use Edward's method for SystemC because processes do not necessarily satisfy this constraint which is not part of the standard SystemC guidelines. Moreover, while the method can theoretically determine the optimal schedule, its complexity is exponential and heuristics are used to provide a schedule in reasonable time. On the other hand, the method highlights that the more accurate the dependency graph, the easier to determine the proper order of signal computation. The recently proposed Liberty simulation environment [8, 6] for microarchitecture exploration implements Edwards's model and scheduling algorithm.

Acyclic scheduling. In practice, we found that providing enough additional dependency information (68 for our supercalar processor simulator) between signals is enough to obtain an acyclic graph, because the natural flow-oriented structure of processor and system architectures already give a fairly (if not fully) acyclic graph. And if the graph is acyclic, Edwards's or Hommais's methods are superfluous because the schedule becomes trivial, i.e., it is given by the graph itself. Therefore, we have developed a simple scheduling technique that is intermediate between static and dynamic scheduling called *acyclic scheduling*. Unlike the

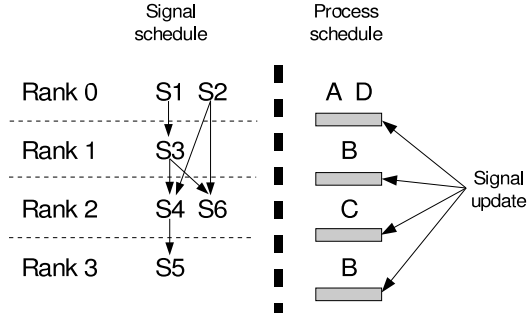


Figure 5. Ranking algorithm.

abovementioned scheduling techniques, we do not attempt to find and schedule strongly connected components, we bluntly interpret the graph as one or several *acyclic graphs*. For that purpose, we simply rank the vertices starting with the source vertices, i.e., the vertices with no inbound edge. Consider again the example of Figure 3. The dependency graph is shown on the left of Figure 5: signals S1 and S2 have rank 0 because they are source vertices; signal S3 has rank 1, signals S4 and S6 have rank 2 and signal S5 has rank 3. The corresponding process schedule is obtained by replacing the signals with the processes updating the signals, as shown on the right of Figure 5.

If the graph has cycles, then we arbitrarily break them by ranking differently the vertices in the cycles. Then, we simply schedule processes according to their rank. If the graph has several components, we will have several acyclic graphs, and we schedule simultaneously processes with the same rank. In summary, we assume the original graph is acyclic, we “propose” the corresponding static schedule to the SystemC engine for the first delta-cycles. If the graph has cycles and if the list of active signals is not empty when all statically scheduled processes have been executed, the SystemC engine resorts to its standard dynamic scheduling. Moreover, the process is robust: if the programmer does not provide enough dependency information, or provide incorrect ones, the dynamic scheduling mechanism will catch up static scheduling errors; performance improves gracefully with the amount of dependency information.

In practice, adding dependency information simply amounts to specifying data dependencies between signals. For instance, in the example of Figure 3, if signal S1 is connected to input port `in1` and signal S3 is connected to output port `out3`, then we simply add the source line `out3(in1)` in the module constructor. For the superscalar simulator with 15000 source lines, we only had to add 68 lines for specifying dependency information. Moreover, such dependency information can be easily extracted by a compiler, so we intend to generate it automatically in the future.

Implementation. A straightforward implementation of the above method is to call statically scheduled processes in the order specified by their rank, updating signals at each step (for each rank value). The advantage of this approach is that no compilation is required, the static schedule is created in the initialization step of the simulation. However, because all statically scheduled processes may not need wake-up each simulated cycle, we had to add a flag in `sc_method` which states whether the input signals of a process have changed (this flag is set by `sc_signal`) in order to avoid useless wake-ups. We found that the need to check each statically scheduled process every delta cycle slows down the central SystemC engine loop and eliminates almost all benefits of reducing the number of process wake-ups (see the code in Figure 6). More precisely, the problem is that the behavior of the conditional branch associated with the `if` instruction of this code has a very irregular behavior. Therefore, the branch predictor of the processor on which SystemC is run is unable to predict correctly and performance strongly degrades, even though the experiments were run on a Pentium 4 2 Ghz with a fairly modern branch predictor.

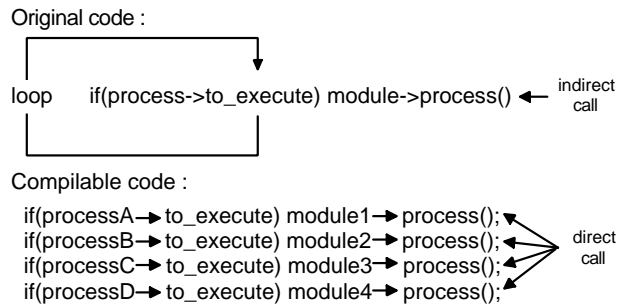


Figure 6. Central SystemC loop: non-compiled and compiled versions.

Therefore, to eliminate this bottleneck, we resorted to a compiled version of the scheduler where the loop over the list of statically scheduled processes is replaced by an unrolled version where each process is directly called. The original single branch is replaced with multiple branches, one per process, so that the branch predictor can much more easily predict their behavior. With this implementation, we found we were able to translate the reduction of the number of process wake-ups into increased simulation speed.

Experimental results. The 5-stage RISC processor has 11 sequential processes and 18 combinational processes. We found that its original graph has no cycle, so that adding dependency information was not necessary to improve its performance, see Figure 7. Using acyclic scheduling, the number of process wake-ups decreased by 22.7% compared to SystemC 2.0.1.

The superscalar processor simulator has 25 sequential processes, 29 combinational processes, and 460 signals. Unlike the RISC processor, its graph has several cycles, but we found that adding 68 signal dependency informations would remove all cycles. The resulting dependency graph has 460 vertices and 1248 edges. Using acyclic scheduling, the number of process wake-ups decreased by 29.9% compared to SystemC 2.0.1. More important, we examined the distribution of the number of process wake-ups per process and per simulated clock cycle, see Figure 8, where a bar at position x represents the fraction of total process wake-ups that corresponds to processes waken up x times within the same simulated clock cycles. For the combinational processes, we found that our acyclic scheduling eliminated almost all process wake-ups in excess, meaning the performance of simulators executed with this schedule can be much more predictable and regular than with a dynamic scheduling algorithm.

The overall speedup of the acyclic scheduling technique is 3.56 for the RISC processor and 1.96 for the superscalar processor, see Figure 7; we can note that the compiled version (see *Acyclic Scheduling*) performs way better than the non-compiled version.

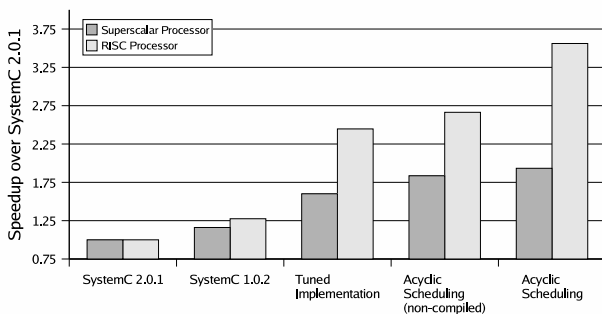


Figure 7. Speedup over SystemC 2.0.1.

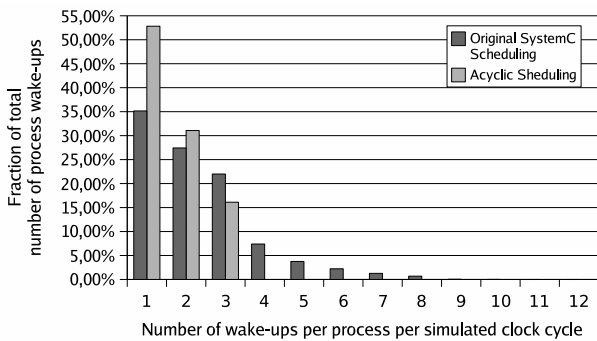


Figure 8. Wake-up distribution for the combinational processes of the superscalar processor simulator.

4 Conclusions and Future Work

While SystemC provides a complete framework for cycle-level architecture simulation and is becoming a *de facto* standard, its execution speed is rather poor. In this article, we showed that this poor performance is due in part to a simulation engine that is not tuned for performance, and to a rather basic scheduling algorithm. We introduced a novel implementation of the SystemC engine fitted with an improved scheduling mechanism that outperforms SystemC 2.0.1 by 1.93 to 3.56 on our two test simulators. Our scheduling algorithm, called *acyclic scheduling* takes advantage of the nature of the signal graphs corresponding to the implementation of processor and system architectures, more precisely their relatively low number of graph cycles (within a simulated clock cycle); this scheduling approach is fairly simpler than previously proposed static scheduling techniques. The only drawback of our approach is that the programmer must add some dependency information to get the best possible performance, even though the approach is robust enough to perform accurately, and even efficiently for some simulators, without such information. While we show that this additional information corresponds to a tiny fraction of the simulator development effort, in the future, we intend to generate it automatically using a preprocessor so as to remove any programmer overhead.

References

- [1] L. Charest, C. Pilkington, and P. Paulin. SystemC performance evaluation using a pipelined DLX multiprocessor. In *DATE '02*, Paris, France, March 2002.
- [2] S. A. Edwards. The specification and execution of heterogeneous synchronous reactive systems, PHD. Thesis. University of California, Berkeley, 1997.
- [3] J. Hennessy, N. Jouppi, F. Baskett, and J. Gill. MIPS: A VLSI processor architecture. In *The Proceedings of the CMU Conference on VLSI Systems and Computations*, Rockville, Md. USA., October 1981.
- [4] D. Hommais and F. Pétrot. Efficient combinational loops handling for cycle precise simulation of system on chip. In *24th Euromicro*, Vasteras, Sweden, August 1998.
- [5] OSCI. SystemC. <http://www.systemc.org>, 2000-2002.
- [6] D. A. Penry and D. I. August. Optimizations for a simulator construction system supporting reusable components. In *Proceedings of the 40th Design Automation Conference*, Anaheim, California, USA., June 2003.
- [7] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), June 1972.
- [8] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with liberty. In *the 34th Annual International Symposium on Microarchitecture*, Austin, Texas, USA., December 2001.