Fast and Modular Transaction-Level-Modeling and Simulation of Power and Temperature

Claude Helmstetter, Tayeb Bouhadiba, Matthieu Moy and Florence Maraninchi

Verimag Research Report n^o

January 15, 2014

Fast and Modular Transaction-Level-Modeling and Simulation of Power and Temperature

Claude Helmstetter, Tayeb Bouhadiba, Matthieu Moy and Florence Maraninchi

January 15, 2014

Abstract

How to cite this report:

```
@techreport {,
   title = {Fast and Modular Transaction-Level-Modeling
and Simulation of Power and Temperature},
   author = {Claude Helmstetter, Tayeb Bouhadiba, Matthieu Moy and Florence Maraninchi},
   institution = {{Verimag} Research Report},
   number = {},
   year = {}
}
```

stimating power consumption and temperature of systems-on-a-chip has become a key point. Early estimations require high-level models and sufficiently fast simulations. We consider the approach called *Transaction-Level Modeling (TLM)*, implemented with the standard SystemC. It enables the execution of the actual embedded software on a simulated hardware. The hardware model mimicks its functional behavior, and can be instrumented to provide timing and consumption estimations. Temperature estimation can be obtained by coupling this SystemC/TLM model with a temperature model. All this allows capturing the effects of the software on power consumption and temperature, e.g., when the power and temperature management policy of the chip is implemented in software (reading temperature sensors, and acting dynamically on the voltage and frequency).

Following existing proposals on the modeling of power in SystemC models, we provide libraries for a systematic, easy, and fast instrumentation of SystemC/TLM approximately-timed models. We adapt the existing instrumentation principles in order to benefit from SystemC/TLM optimizations like temporal decoupling and direct-memory-interface, which are compulsory in industrial-size models.

We evaluate our approach on several use cases, showing that after a simple integration, the software developer can easily distinguish an ill-founded from a correct software in terms of power consumption or temperature control.

1 Introduction

Reducing power consumption of Systems-on-a-Chip is an important challenge. Clearly, portable devices should save energy to maximise battery life, but other issues like heat dissipation, voltage drop or faster ageing due to overheating are also of growing importance.

With modern CMOS processes, static power consumption, due to leakage current, is increasing. Power-saving techniques like clock-gating, that act only on dynamic consumption are no longer sufficient. More heavy-weight techniques are necessary to control the consumption. A system-on-chip may be structured into several *power or frequency domains*, so that the clock and power supplies of a part can be controlled independently of the others, based on sensors for temperature, voltage, ... The power-management policy itself is usually implemented in software.

Developing such power-management policies requires an execution platform, including models of sensors, actuators, and taking into account the feedback loop between actuators and sensors. Low-level (gate or RTL) simulations are possible, but too late in the design flow and their simulation is not fast enough for system-level simulation including non-trivial software. On the other hand, an accurate model of power consumption and temperature requires a realistic model of timing. SystemC/TLM allows several levels of details for the timing: cycle-accurate is the most precise, while “approximately-timed” [2] is more relaxed and improves simulation speed.

Instrumentation guidelines to introduce power and/or temperature estimations into SystemC models at the “approximately-timed” transaction-level or at the cycle-accurate level, have been proposed in [5, 6, 12, 18].

The instrumentation consists in associating physical parameters to the hardware behavior described in SystemC/TLM. The simulation then computes the consumption and temperature of the whole system-on-a-chip, based on these per-component parameters, including the effect of power and thermal management policy and software. This allows detecting *non-functional* bugs in the embedded software (failure to enter a low-power mode, polling instead of explicit wait for an interrupt) or comparing different policies. The contributions of this paper are:

- We propose a general instrumentation method based on *activity ratios*. The power consumption of each component is then computed as a generic function of the voltage, the frequency, the temperature, and this component’s activity ratio. Modelling dynamic voltage and frequency scaling (DVFS) requires no extra modelling effort, and the generic function is provided in the library.
- We provide new modeling guidelines, and new implementations of the principles described in [5], which are now compatible with existing TLM optimisation techniques, like *temporal decoupling* and *direct memory interface* (DMI). This is compulsory to get satisfactory simulation speed, but requires a significant modification of the principles. In contrast with the results reported in [5], the simulation speed overhead of our instrumentation is independent of the SystemC step.

- We report on several new use cases to show how the simulation can help software developers.

We use ATMI [13] for temperature modeling and simulation, and we develop a graphical interface (Fig. 3) showing the evolutions of the temperature map of the circuit.

2 Related Work

The TLM Power tool presented in [12, 18] proposes to model a system-on-a-chip in SystemC/TLM, running the actual software on top of a simulated hardware. The objective is to validate a power-management policy. It is based on the same *power-state* models as described in [3, 4].

The instrumentation principles described in [5] use the same principles as [12, 18] for the power model, augmented with the potential feedback of temperature on the functionality of the chip, and on the static power consumption.

All the abovementioned solutions include all hardware components, at a level of details sufficient to run the actual software. As a consequence, they focus on simulation: exhaustive exploration with model-checking like [7] is not applicable at this level of details. In [11], a system-level *analytical* model is proposed, to capture the consumption and thermal behaviour of a chip with *Power Variability Curves*, based on the framework of real-time calculus. The actual software is not executed, but modelled as a set of tasks, and the hardware architecture is not detailed. Power consumption is considered to be a function of the executing software task, hence only processors are considered. [11] computes guaranteed bounds on temperature peaks. The method is intended to be used at a very early stage of the design.

The work described in [15] is a system-level simulation method including the functionality and power consumption. It focuses on instruction-level power consumption for software execution, describing in details how to get the parameters of the power-model, with measures. It is limited to power-consumption and does not take temperature into account. A case-study from Intel is described, and the simulation results compared to measures on the real chip.

3 New Power and Temperature Modelling Method

3.1 Initial SystemC/TLM Models

A SystemC/TLM model is made of components and connections. Each component describes the functional behavior of a hardware element (a processor, a bus, a memory, ...) and its approximate timing. The model of the processor is able to run embedded software, for instance using an instruction-set-simulator. Figure 1 is an example with a processor, a memory, a temperature sensor, a DVFS controller, an interrupt controller, a VGA screen controller, and a bus. The simulation is performed by a discrete-event engine with simulated time.

3.2 Instrumentation Parameters

Instrumenting such a model for power consumption estimation means associating instantaneous power consumption with *states* of the components (e.g., Off, Idle, ...), or with activities (e.g., low or high traffic on the bus). This is done by function calls in the SystemC code that constitutes the model of the hardware. For each hardware component of the SystemC/TLM initial model, we now define the following categories of parameters:

- **static parameters:** *Floorplan parameters:* location of the component on the chip; used to compute the area of the component, and the neighbour relations. *Technology-dependent parameters:* physical values that depend on the technology used to produce the physical chip; this includes capacitance, leakage current, influence of temperature on the leakage, etc. Most of these values are common to all components.
- **dynamic configuration parameters:** *Voltage* used by the component. May change from time to time if the chip provides voltage scaling. *Frequency* used by the component. May change from time to time if the chip provides frequency scaling. This value is used to compute the clock period.

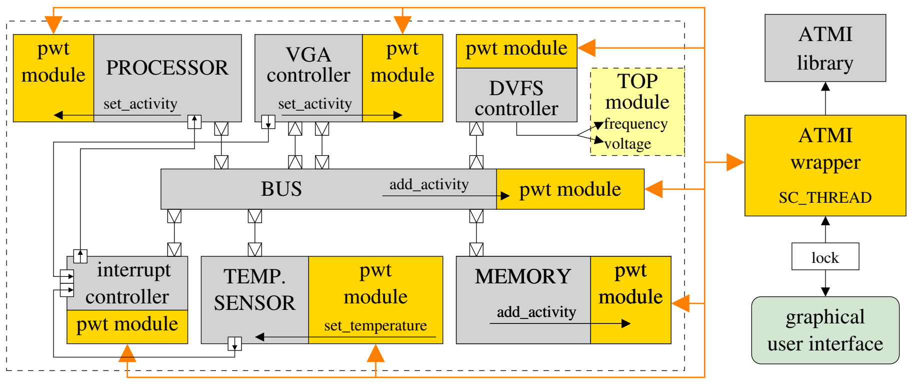


Figure 1: Example of a minimal SoC model with its power and temperature extensions.

```
while (true) {
1.  wait(atmi_step_duration);
2.  for each component, compute its average power density
3.  during the last step elapsed.
4.  atmi_simulator_step(atmi_instance,
5.                        power_densities) //call ATMI
6.  set component temperature and call associated callbacks
}
```

Figure 2: SystemC ATMI wrapper.

- **activity ratio:** approximate ratio of gates that are active. This is a number between 0 and 1: close to 1 means the module is busy, below 0.1 generally means the component is idle. This value can change up to every clock cycle.

In order to relate the functional behavior of a system-on-a-chip, its timing and power consumption, and its temperature, we map the SystemC/TLM components to the areas of an ATMI [13] temperature model. ATMI takes as input a *floorplan*, i.e., a set of areas of the chip, described by their coordinates, and the initial temperature.

4 Instrumentation Method and Architecture of the Tool

4.1 Overview

Figure 1 shows an example model of system-on-a-chip, and the main elements of our solution. The left part is the SystemC/TLM model. It communicates with ATMI through the ATMI wrapper. A graphical user interface (GUI) can be used to monitor and control the simulation, see Fig. 3.

During simulation, ATMI computes the temperature of each area based on its power consumption, expressed as a power density, and computed on the SystemC side (thanks to the `pwt_module`, see below). This computation is done at a regular pace, such as once every millisecond (SystemC time). Since ATMI is packaged as a C library, it can be directly called from SystemC code. This is done by the ATMI wrapper, written in SystemC. The wrapper contains a SystemC thread that calls the ATMI library repeatedly, according to the ATMI pace, as shown on Figure 2. Given the power densities, computing the component's temperature is just a function call to the ATMI library. For any SystemC component, it is possible to provide a callback method that is called each time the temperature of the corresponding area changes according to ATMI. For example, the temperature sensor defines a callback method that raises an interrupt when the temperature reaches some thresholds.

4.2 Instrumentation on the SystemC side

The instrumentation of a component is done by inserting calls to a function `set_activity` and `add_activity` (more details in section 4.4). Each SystemC component mapped to an ATMI area must inherit from the `pwt_module` class (`pwt` standing for *PoWer and Temperature*). This class stores all the static and dynamic parameters, and the activity ratios (as defined in section 3.2). In order to define frequency and power domains, we allow additional PWT modules which do not correspond to actual hardware components, i.e., are not mapped on the floorplan. Such modules only provide the voltage and frequency parameters, which are forwarded to their children modules. Other methods are disabled; in particular, they have no power densities. For example, the chip on Figure 1 has one DVFS (*Dynamic Voltage and Frequency Scaling*) controller and a single power domain. So, in SystemC, the model of the DVFS controller is bound to the top module; the DVFS controller TLM module calls the methods `set_frequency` and `set_voltage` of the top module, which in turn calls the `set_frequency` and `set_voltage` methods of all its children PWT modules.

4.3 Computation of Power Densities

The wrapper (Figure 2) computes the the average power density consumed during the last step elapsed (lines 2-3). This computation is done in the new `pwt_module` class. Given all the parameters of the component (static, dynamic, and activity ratios), the `pwt_module` class computes the average of the power density during the last step elapsed. It is the sum of the static power and the dynamic power.

The *static* power is due to the leakage current, and it is proportional to the voltage and the leakage current intensity. The intensity itself increases when the temperature increases; in the current implementation, we use a linear approximation of the temperature effect.

The *dynamic* power corresponds to the cost of voltage changes in gates. It is proportional to the frequency, to the number of gates involved (i.e., the activity ratio), and to the square of the voltage. Moreover, it is proportional to a constant that depends on the capacitance per gate and on the gate density. The general formula is of the form:

$$P = P_{static} + P_{dynamic} =$$
$$V \times K_1 \times (1 + K_2 \times T) + F \times V^2 \times \alpha \times K_3$$

where V is the voltage, T is the temperature, F is the frequency, α is the activity ratio, and K_i are static parameters depending on the component area and on the technology.

Because PWT modules contain the general formula, which involves explicitly the frequency and the voltage, the power model manages DVFS for free. Other approaches [12] let model developers provide the actual power value, using their own function that may or may not take into account that the voltage or the frequency may change. If needed, a module that has a specific power model can also redefine the method that computes its power density.

4.4 Setting the Activity Ratio

The main task to extend a SystemC/TLM model with power consumption and temperature estimations is to set dynamically the activity ratio of each module. The `pwt_module` class provides two ways to set this ratio. The first one allows to set the activity ratio starting from *now* and until another level is set:

```
void set_activity(float ratio,
                                                    sc_time now = sc_time_stamp()).
```

The second one adds some extra-activity spanning on an interval of some duration (given as a number of cycles), starting from *now*:

```
void add_activity(float ratio_increment,
                  unsigned nb_cycles,
                                                    sc_time now = sc_time_stamp()).
```

In general, the first method is best suited for initiator modules (like processors) whereas the second is better for interconnects (busses) and target modules (like memories). For example, a processor's TLM model will call the `set_activity` method when it enters an *idle* state and when it becomes *busy* again. Using this method, we get an activity state-based power model, as in [12]. If more accuracy is needed, one can develop an instruction-based power model as in [8] by using the second method and calling add_activity for each instruction, with a ratio depending on the instruction kind and the register values. Obviously, the second approach requires a lot of additional manpower and will slow down the simulation.

Concerning interconnect and target modules, the best solution is to call the add_activity method once per transaction. In general, the *ratio increment* depends on the command (READ or WRITE) whereas the duration (nb_cycles) depends on the transaction size. Note that it would be harder to use the activity state-based method, because the local activity state depends on the external initiators, and is not known locally.

The add_activity method could be implemented using two calls of the set_activity method, as follow: remember the current activity level as current_ratio, set the activity level to current_ratio + ratio_incr, increase the local date, and finally re-set the activity to current_ratio. However, this code is not reentrant, which is mandatory for a bus or memory module, and pretty slow. On the contrary, the add_activity method is reentrant and quick.

The rationale of the parameter "sc_time now" of the methods set_activity() and add_activity is to ensure the compatibility with temporal decoupling [16], when using the coding rules defined in [2]. When temporally decoupled, the local date of a process is expressed as "sc_time_stamp()+local_offset" (instead of "sc_time_stamp()"), allowing to advance the local time by executing a low-cost "local_offset += T" instead of a costly "wait(T)". This local offset is part of all transactions, so it can be used by interconnects and target modules too. Consequently, to set the activity ratio at the right date, the methods set_activity and add_activity must be called with the parameter now set to "sc_time_stamp()+local_offset". Since this parameter has a default value, it can be safely ignored for all processes that are not temporally decoupled.

The date used to call the activity methods may be later than the next ATMI step boundary. Indeed, some TLM modules modelled at a coarse grain may simulate up to one second of SystemC time before yielding back to the scheduler. Consequently, each PWT module contains a list of activity counters. The list head contains the activity counter of the current ATMI step, and successive list elements store the activity of future steps. The ATMI wrapper pops the front element once every step. The traffic model of [5] can be implemented on top of this, but it is currently not part of the tool presented here.

4.5 Direct-Memory-Interface Management

To improve simulation speed, some TLM modules use a technique called *Direct Memory Interface* (DMI). It speeds up memory accesses by providing the initiator (e.g., a processor given as an instruction-set-simulator) with a pointer to the memory array. So, when accessing memory, the initiator will directly use the memory pointer instead of generating a transaction that goes through the bus. Since a transaction would involve many indirect function calls plus routing in the interconnects, the speed gain is significant. That is functionally correct but may bypass some side effects, because the code related to timing and power into the interconnects and the memory is no longer executed. For the timing issue, [2] suggests to provide the initiator with two durations: the read latency and the write latency. Thus, the initiator can add the latency to its local offset when a memory access is simulated. Because the latencies depend on the frequency, the *DMI descriptor* must be updated every time the frequency is changed. We use the same idea for power modelling. However, providing a single activity ratio increment per transaction is not enough, since the activity increment must be added to each module involved in the transaction. Indeed, if the additional activity was assigned to the initiator, then the initiator temperature would be overestimated whereas the bus and memory temperatures would be underestimated. The solution is to add into the DMI descriptor a pointer list of all PWT modules that are on the transaction path. Our DMI manager class provides a method apply_side_effects(command, size) that increases the local time offset according to the latency and calls the add_activity method of all the PWT modules in the list.

For monitoring, the GUI provides the current temperatures as text values, a graph of the floorplan where each module is coloured with respect to its temperature (from blue for cold, to red for hot module). Additionally, a plot provides either the temperature or the power consumption or the power density of each module depending on the time.

5 Evaluation and Applications

5.1 Development Cost

For the development and the evaluation of our approach, we have developed a demonstration platform based on a small FPGA system. The main components are a processor (MicroBlaze) and a VGA controller. There are two memories, one for instructions and the other for data, plus the usual devices: timers, UART, interrupt controller, etc. Compared to the initial FPGA system, we have added in the TLM model a temperature sensor and a DVFS controller. The whole TLM model uses the blocking TLM interfaces of [2], with the generic payload plus an ignorable extension for DMI configuration. We reuse some open-source TLM code from SoCLib [1] and SimSoC [9].

We have applied some classic optimisations in the TLM model, so that the base simulation speed is similar to the simulation speed of an industrial TLM model. In particular, we use *temporal decoupling* in all places where it is useful, and the processor and VGA controller models use the *Direct Memory Interface* mechanism. When the processor is busy, the simulation speed is around 50 MIPS (million instructions per second).

The TLM model without power and temperature counts 5000 lines of code, and uses some general development kits counting in total 1400 lines of code. In this version, the temperature sensor and the DVFS controller modules are included but they have no behaviour.

The core classes we have developed for power and temperature modelling counts 700 lines of code (not including the ATMI library, which is 2700 lines long). Additionally, the graphical user interface counts close to 600 lines.

For the instrumentation of the platform itself, we have added about 100 lines of code. Note that this is quite small compared to the platform size, showing that once the tools are available, instrumenting an existing TLM model for power and temperature estimations requires a very little cost. One must provide the physical values used in the power and thermal model; this calibration task is out of the scope of this paper.

5.2 Simulation Speed Overhead

To evaluate the simulation speed, we use our demonstration platform and make it run a benchmark application. In this benchmark, the processor is periodically computing: it waits one second and then computes during about 0.8 seconds (SystemC time). The application computes the "game of life", waiting 1 second between images. Additionally, the VGA controller is active and loads the image buffer 60 times per second. Between two consecutive reloads, the VGA controller remains idle during a few milliseconds.

The first time the model is simulated, the ATMI library computes many data in advance in order to accelerate the simulation itself. Those data are cached in a file for future simulations. Modifying the floorplan or some technology-dependent parameters requires to compute this file again. This computation takes about two minutes.

Simulating 10 seconds (SystemC time) takes:

- 3.4 seconds (wall-clock time) for the initial functional TLM model
- 6.4 seconds with power and temperature estimations (6.6 seconds with GUI), assuming that the ATMI cache file was ready.

So, the simulation duration overhead for instrumented models is about **+88%**. We consider that it is a significant but acceptable overhead.

If the DMI is disabled, the functional simulation consumes 9.1 seconds whereas the PWT simulation consumes 12.8 seconds. So, without our extended DMI mechanism, the total overhead would be 9.4 seconds, i.e. +276% (5.7 seconds for disabling the DMI plus 3.7 seconds for power and temperature computations). Naturally, running the PWT simulation with DMI but without the extension is quick ($\approx$5 seconds) but incorrect: we have observed errors of more than 1 degree Celsius.

Looking at the profile obtained with `callgrind+kcachegrind` [17], we notice that there are two performance-consuming spots: 1. computations internal to the ATMI library ($\approx$28% of total simulation time), 2. applications of transaction side effects when using DMI ($\approx$12% of total simulation time). The `pwt_module` class has been implemented with the optimisation of this second performance-consuming spot in mind.

Concerning the time spent in the ATMI library, the user may optimise it by adapting the ATMI step duration. The values above are given for a step duration of one millisecond. As shown by Table 1, the longer the ATMI step, the faster the simulation, but the cost is a loss of accuracy. The temperature error is higher in modules whose power density changes at a fast pace.

Table 1: Effect of the ATMI step duration

ATMI step	simulation duration	standard deviation (σ)			
		processor	VGA	bus	temp. sensor
0.25 ms	15.3 s (+139%)	*reference*	*reference*	*reference*	*reference*
0.5 ms	8.9 s (+39%)	0.01 °C	0.05 °C	0.09 °C	0.00 °C
1 ms	6.4 s (*ref.*)	0.03 °C	0.11 °C	0.12 °C	0.01 °C
2 ms	5.2 s (-19%)	0.06 °C	0.23 °C	0.15 °C	0.03 °C
4 ms	4.6 s (-28%)	0.13 °C	0.42 °C	0.19 °C	0.05 °C

5.3 Applications

Using the "game of life" benchmark previously presented, we can observe that temperature plots are as expected. Figure 4 shows those plots at different time scales. Looking at a short time range, we see that the VGA temperature fluctuates with an amplitude slightly above 1 °C; as a consequence, the temperature sensor fluctuates too, but with a smaller amplitude. On the second plot, we see that the processor temperature fluctuates at a slower pace, since it is computing about one second every two. Moreover, other module temperatures evolve according to the processor temperature. Finally, the third plot shows that the whole system takes about 100 seconds to reach its maximum temperature. It is one reason why simulators must be fast enough to allow simulations of many minutes of SystemC time.

One possible application of our tool is to detect non-functional errors in embedded software, such as polling a device register instead of using idle mode and interrupts. Figure 7 shows what happens if the previous benchmark uses polling instead of interrupts. The functional behaviour is exactly the same, but we immediately see that the temperatures keep increasing and that the real chip would overheat. Note that the bus temperature is higher during polling than during frame computation due to the high polling traffic.

Another application is the development and validation of the voltage and frequency control. One simple solution to avoid overheating is to switch between two modes: a default fast mode where frequency and voltage are high, and a backup low-power mode where voltage and frequency are low. The controller (i.e., a part of the embedded operating system) programs the interrupts of the temperature sensor module according to two thresholds: the high threshold is used to avoid overheating and causes the switch to the low-power mode, whereas the low threshold determines when to switch back to the fast mode. Testing this algorithm on a pure functional TLM model is quite difficult, because temperature sensor interrupts will not occur or will occur at unrealistic dates. On the contrary, using the extended TLM model and its GUI allows to check easily the controller behaviour, as shown on Figure 5. Again, note that the simulation must be at least 20 seconds long to be useful, so simulation speed matters.

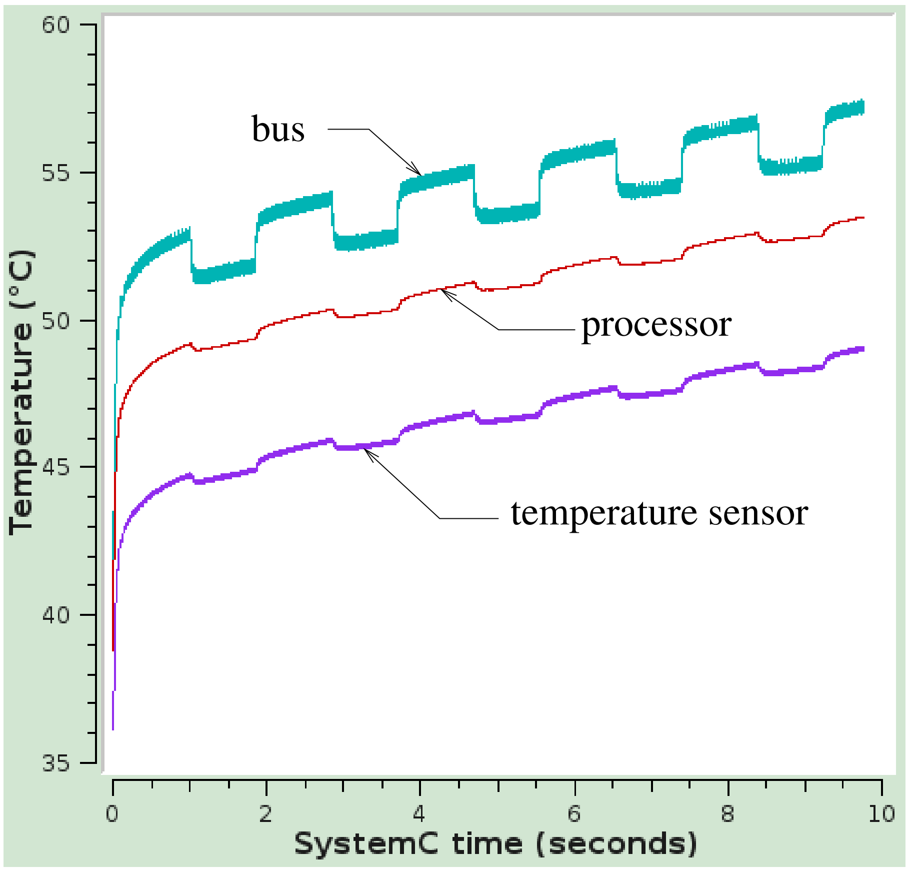


Figure 7: Variant of the "game of life" benchmark using polling.

Such temperature control based on low and high thresholds has some drawbacks; one is that the frequent temperature changes may raise the failure rate of the system [19]. Another approach is to use a PID controller. We have implemented this approach in the embedded software of our demonstration platform. As shown by Figure 6, we see that the PID-controlled temperature curve is smoother than the previous threshold-controlled temperature curve. The first plot shows a simulation with a badly tuned PID controller, where the VGA controller temperature oscillations are amplified instead of smoothed, meaning that the gain parameters are likely too high.

6 Conclusion

We have shown how to modify existing proposals for the power and temperature instrumentation of SystemC/TLM models, so as to make them compatible with state-of-the-art transaction-level modeling guidelines and optimizations. In particular, our solution can benefit from temporal decoupling and direct-memory-interface, which is compulsory to get reasonable simulation times, even for the non-instrumented models. The simulation speed overhead of the instrumented models is therefore kept as small as possible. We have proposed to instrument models with *activity ratios* instead of power values, which makes the models more generic, in contexts where the frequency or the voltage may change. We have shown use cases where the approach presented can help the embedded software developer to detect issues related to power consumption or temperature.

Using ATMI limits us to 2D designs. It would be interesting to test other thermal solvers such as HotSpot [10] or 3D-ICE [14], which are able to manage 3D chips.

Further work includes the development of calibration tools: assuming that the temperature or power consumption are known but not some activity ratios or physical parameters, a calibration tool would have to find the best values for unknown figures.

Another interesting feature would be to add random noise either to the power densities (corresponding to model inaccuracies) or to the computed temperature (to model the inaccuracy of actual hardware temperature sensors). Indeed, an illusion of accuracy could mislead the software developer by favouring temperature management policies that are not robust to inaccuracies.

References

[1] An open platform for virtual prototyping of multi-processors system-on-chip. `http://www.soclib.fr/`.

[2] Accellera Systems Initiative. *IEEE 1666 Standard: SystemC Language Reference Manual*, 2011.

[3] L. Benini, R. Hodgson, and P. Siegel. System-level power estimation and optimization. In *Proceedings of the 1998 international symposium on Low power electronics and design*, ISLPED '98, pages 173–178, New York, NY, USA, 1998. ACM.

[4] R. A. Bergamaschi and Y. W. Jiang. State-based power analysis for systems-on-chip. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 638–641, New York, NY, USA, 2003. ACM.

[5] T. Bouhadiba, M. Moy, and F. Maraninchi. System-level modeling of energy in TLM for early validation of power and thermal management. In *Design Automation and Test Europe (DATE)*, Grenoble, France, Mar. 2013.

[6] T. Bouhadiba, M. Moy, F. Maraninchi, J. Cornet, L. Maillet-Contoz, and I. Materic. Co-Simulation of Functional SystemC TLM Models with Power/Thermal Solvers. In *Virtual Prototyping of Parallel and Embedded Systems (VIPES)*, Boston, États-Unis, May 2013.

[7] D. Das, P. P. Chakrabarti, and R. Kumar. Thermal analysis of multiprocessor SoC applications by simulation and verification. *ACM Trans. Des. Autom. Electron. Syst.*, 15:15:1–15:52, March 2010.

[8] N. Dhanwada, I.-C. Lin, and V. Narayanan. A power estimation methodology for systemc transaction level models. In *Proceedings of the 3rd*, CODES+ISSS '05, pages 142–147, New York, NY, USA, 2005. ACM.

[9] C. Helmstetter, V. Joloboff, and H. Xiao. SimSoC: A full system simulation software for embedded systems. In *Open-source Software for Scientific Computation (OSSC)*, pages 49–55, 2009.

[10] W. Huang, S. Member, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, M. R. Stan, S. Member, and S. Member. Hotspot: A compact thermal modeling method for CMOS VLSI systems. *IEEE Transactions on VLSI Systems*, 14:501–513, 2006.

[11] P. Kumar and L. Thiele. System-level power and timing variability characterization to compute thermal guarantees. In *CODES+ISSS 2011*, pages 179–188, Taipei, Taiwan, 2011. ACM.

[12] H. Lebreton and P. Vivet. Power modeling in SystemC at transaction level, application to a DVFS architecture. In *Symposium on VLSI. ISVLSI'08.*, pages 463–466. IEEE, 2008.

[13] P. Michaud and Y. Sazeides. ATMI: analytical model of temperature in microprocessors. *Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2007.

[14] A. Sridhar, A. Vincenzi, M. Ruggiero, T. Brunschwiler, and D. Atienza. 3d-ice: Fast compact transient thermal modeling for 3d ics with inter-tier liquid cooling. In *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, pages 463–470, 2010.

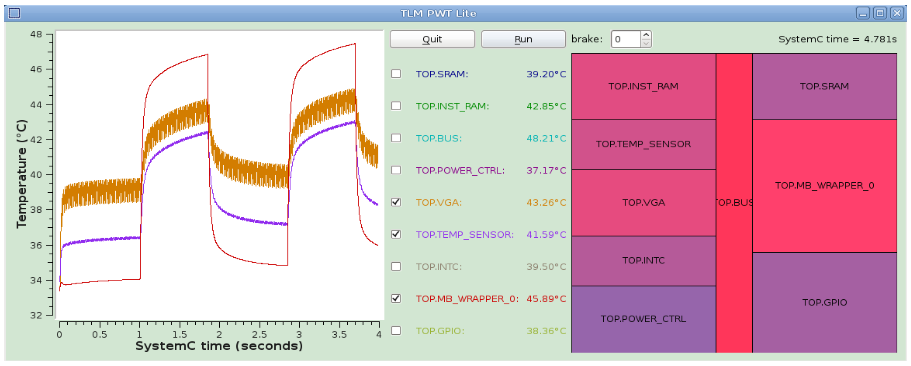
[15] A. Varma, E. Debes, I. Kozintsev, P. Klein, and B. L. Jacob. Accurate and fast system-level power modeling: An xscale-based case study. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[16] E. Viaud, F. Pêcheux, and A. Greiner. An efficient TLM/T modeling and simulation environment based on conservative parallel discrete event principles. In *DATE'06*, pages 94–99, March 2006.

[17] J. Weidendorfer. Sequential performance analysis with callgrind and kcachegrind. In M. M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Parallel Tools Workshop*, pages 93–113. Springer, 2008.

[18] M. Yasin, C. Koch-Hofer, P. Vivet, and D. Greaves. TLM power 3.0 (CBG) user manual. koo.corpus.cam.ac.uk/tlm-power3, 2012.

[19] F. Zanini, D. Atienza, L. Benini, and G. De Micheli. Multicore thermal management with model predictive control. In *European Conference on Circuit Theory and Design (ECCTD 2009)*, volume 1, pages 90 – 95. IEEE Press, 2009.



brake: 0

SystemC time = 4.781s

☐	TOP.SRAM:	39.20°C
☐	TOP.INST_RAM:	42.85°C
☐	TOP.BUS:	48.21°C
☐	TOP.POWER_CTRL:	37.17°C
✓	TOP.VGA:	43.26°C
✓	TOP.TEMP_SENSOR:	41.59°C
☐	TOP.INTC:	39.50°C
✓	TOP.MB_WRAPPER_0:	45.89°C
☐	TOP.GPIO:	38.36°C

Figure 3: Graphical user interface, implemented in the Qt framework, and running in a distinct Posix thread than the SystemC simulation. The GUI allows to pause the simulation, or to reduce the SystemC simulation speed. It shows curves and the evolving temperature map.

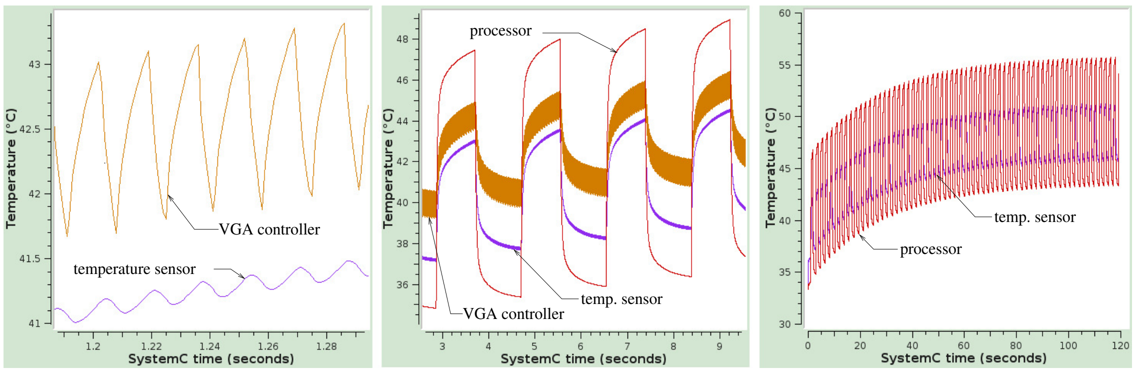


Figure 4: Temperature plots for the “game of life” benchmark, with different time scales.

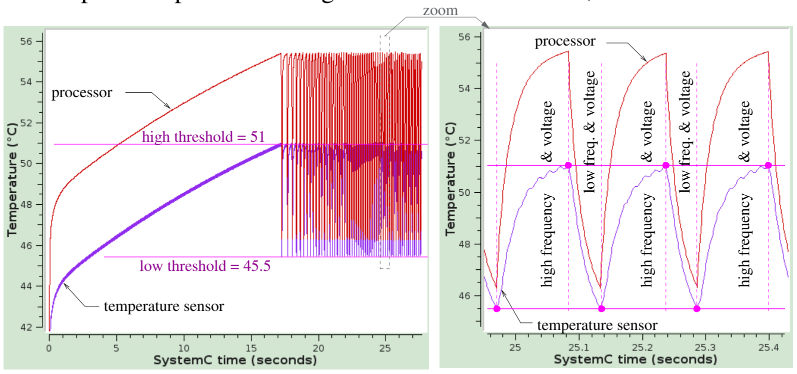


Figure 5: Temperature control with *low* and *high* thresholds: the software controller toggles between low and high power modes according to the temperature

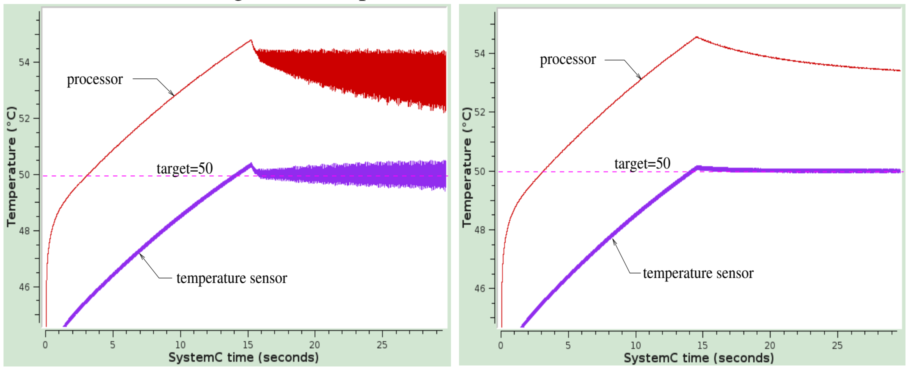


Figure 6: Temperature control using a PID controller, with distinct gain parameters