Fast and Accurate TLM Simulations using Temporal Decoupling for FIFO-based Communications

Claude Helmstetter*‡, Jérôme Cornet†, Bruno Galilée†, Matthieu Moy‡ and Pascal Vivet*
*CEA-Leti, Minatec Campus, Grenoble, France, Email: *firstname.lastname*@cea.fr
†STMicroelectronics, Grenoble, France, Email: *firstname.lastname*@st.com
‡Verimag (UMR CNRS 5104), Grenoble INP, France, Email: *firstname.lastname*@imag.fr

Abstract

A known approach to improve the timing accuracy of an untimed or loosely timed TLM model is to add timing annotations into the code and to reduce the number of costly context switches using *temporal decoupling*, meaning that a process can go ahead of the simulation time before synchronizing again. Our current goal is to apply temporal decoupling to the TLM platform of a heterogeneous many-core SoC dedicated to high performance computing. Part of this SoC communicates using classic memory-mapped buses, but it can be extended with hardware accelerators communicating using FIFOs. Whereas temporal decoupling for memory-based transactions has been widely studied, FIFO-based communications raise issues that have not been addressed before. In this paper, we provide an efficient solution to combine temporal decoupling and FIFO-based communications.

I. Introduction

When designing a new complex SoC, it is now usual to develop an abstract and executable model of the SoC hardware. This model, called *transaction-level (TLM) model* , represents the behavior of the hardware; it may ignore its micro-architecture details (e.g., pipelines, caches, etc) but must be accurate enough to simulate the real embedded software. Such models are generally based on the IEEE SystemC and TLM libraries [1], and, from a software point of view, are collaborative multi-thread programs with a discrete simulated time.

The transactional abstraction level can be subdivided into many *coding styles* according to their timing accuracy, ranging from *untimed* to *cycle-accurate* [1]. Obviously, a better timing accuracy allows the use of the TLM model for early performance evaluations, but unfortunately a better accuracy induces longer development time and slower simulation speed. Thus, a trade-off has to be found according to the use case [2].

We are interested in improving the timing accuracy of *loosely-timed* TLM models while minimizing simulation speed loss and development overhead. To this goal, timing annotations must be integrated into the model. The basic and usual way to add a timing annotation is to insert an instruction telling the simulation kernel to suspend the current process for a given duration (expressed in simulated time). By default, this instruction (e.g., SystemC `wait`) creates a *context switch*, so other processes having a task planned during this given duration can execute before the first process is resumed, thus keeping all processes synchronized together.

Context switches are costly in terms of simulation speed, even with optimized thread libraries. For a finely annotated TLM model, if there is one context switch per timing annotation, then the context switches would become the bottleneck of the simulation speed. The basic idea of *temporal decoupling* is to let process advance their local time in the future, until a synchronization is required [3]. The difficulty is to synchronize only when needed: removing synchronizations always improves simulation speed but may introduce timing inaccuracies or even functional errors.

Temporal decoupling is described in the TLM reference manual [1], and is already used in industrial models. However, the description made in this manual focuses only on memory-mapped communications. Some modern SoCs for high performance computing, such as the industrial case study we will present, are based on a heterogeneous architecture mixing memory-mapped buses and stream-based subsystems based on FIFO protocols. In this work, we have developed and evaluated modeling techniques for applying temporal decoupling to FIFO-based communications. The key idea is to make the model of the hardware FIFO aware of the process local time, in a way that allows to remove most of the context switches without disrupting the timing.

Temporal decoupling is detailed in Section II. Then, Section III describes our solution for applying temporal decoupling to FIFO-based communications. Section IV is dedicated to functional validation and performance evaluation. Finally, section V concludes the paper.

II. Temporal Decoupling

A. Principles

In a temporally decoupled model, each process has its *local date*, which is greater or equal than the global date managed by the simulation kernel. In SystemC, this global date is provided by the function `sc_time_stamp()`. In the sequel, we call `local_time_stamp()` the function that returns the date of the current process. A process is said to be *synchronized* if its local date is equal to the global date.

Any core library for temporal decoupling provides two basic functions: a low-cost `inc(duration)` that increases the local time, and a costly `sync()` that generally requires a context switch. The `sync()` function ensures that other processes have executed up to the caller process local time. The usual implementation of `sync()` executes a SystemC wait of the duration *local date - global date*. Other implementations

978-3-9815370-0-0/DATE13/©2013 EDAA

are possible: [4] shows that `sync()` can be implemented in a distributed way without relying on a global date.

Beyond reducing the number of context switches, temporal decoupling is also useful for SystemC parallelization [4], [5], and interaction with other simulation engines [6].

The TLM reference manual suggests to synchronize the processes according to a *global quantum*, defining the maximum amount of time before a process must synchronize. For example, with of quantum of $Q = 1\,\mu s$, there will be one context switch per simulated μs and per process. Let consider a model where a process simulates a computation, and a second process sends a cancellation message to the first at the date T; the first process may receive the cancellation message when its local date is already $T + Q$, thus introducing a timing error of Q compared to a non-decoupled system. So we see that a large quantum is bad for accuracy, but good for speed. This quantum is generally a runtime parameter, and so temporal decoupling can be disabled by setting it to zero. Using a very large quantum is generally useless with respect to simulation speed, because once the time spent in context switches is small compared to the total simulation time, any further gain in context switches has a negligible effect on the total.

Relying on a global quantum is often insufficient. Consider for example the following code that set a flag for $10\,ns$: `flag=1; inc(10,SC_NS); flag=0;`. Unless the quantum is smaller than $10\,ns$, it is impossible for another process to see that this flag has been set. The solution to keep the expected model behavior is to add an explicit `sync()` before resetting the flag. More generally, all *synchronization points*, as defined in [7], requires a `sync()` statement.

B. FIFO issue

Fig. 1 presents a small model where two processes communicate using a FIFO. The process code is fully annotated with `inc` statements. Implementing a regular FIFO in SystemC without temporal decoupling is relatively easy, and several implementations already exist, like the `sc_fifo` provided with SystemC, or HetSC's KPN model of computation [8]. When temporal decoupling is used, a solution is to take a regular FIFO and to add a `sync()` at the beginning of each public method. Because there will be one context switch per access, this solution is inefficient in term of simulation speed but we consider that it represents the behavior and the timing of the real system as faithfully as possible. Fig. 2 provides a graphical representation of this model execution. Because executing `inc(d); sync()` is equivalent to `wait(d)`, we consider that this model execution does not use temporal decoupling.

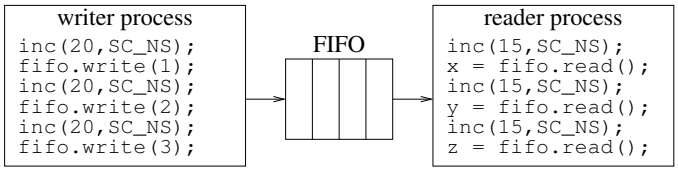

Fig. 1. Basic example using a FIFO and temporal decoupling.

On the contrary, Fig. 3 represents what happens if the `sync()` are all removed. Because the writer changes the state of the FIFO at the global date $t = 0$, the reader executes as if

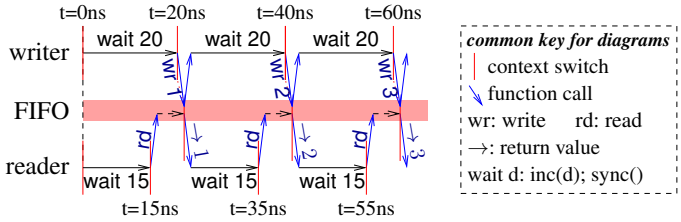

Fig. 2. Execution trace without temporal decoupling.

data were already available (instead of waiting data for 5 ns as in Fig. 2), and so the timing is incorrect. On an example as simple as this one, it looks simple to add a single `sync()` at the right place that could suppress the timing error (e.g., just before the third write). However, any real size model where many processes may be blocked by an empty or full FIFO requires an automatic solution.

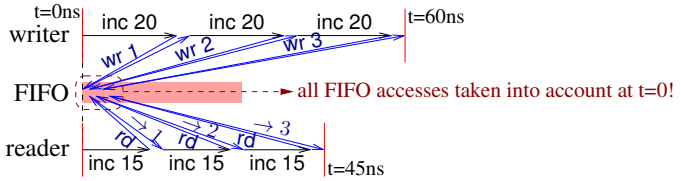

Fig. 3. Execution trace with temporal decoupling but no synchronizations.

III. Smart FIFO for Temporal Decoupling

The Smart FIFO implements three interfaces, as described by Fig. 4. The *read-side* and *write-side* interfaces provide blocking read and write accesses, with additional methods and events for simulating non-blocking accesses. The Smart FIFO assumes that each side is always accessed by the same process; if it is not the case in the design, then an arbiter must be added to ensure that two successive accesses on the same side cannot have decreasing local dates (i.e., time must go forward on each side, but no ordering with the other side is required). Note that we use a map to associate SystemC process handles with their local date, thus avoiding to communicate this date explicitly.

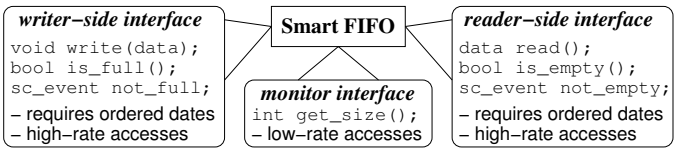

Fig. 4. The Smart FIFO interfaces.

Our main case study is a SoC with hardware accelerators communicating through FIFOs. These accelerators are controlled by some embedded software that must be able to monitor the accelerators and their FIFO; knowing the FIFO filling levels can be used for debug and dynamic performance tuning. That is the rationale for the third *monitor* interface.

The goal is to obtain exactly the same timing with the Smart FIFO compared to an execution without temporal decoupling; all events must happen at the same date, but the schedule and the number of delta-cycle may change.

A. The Smart FIFO: blocking interfaces

Intuitively, the Smart FIFO associates a time stamp with each data item, and uses this time stamp to update the local date of the reader process. That is very similar to how an interconnect manages transactions, and that idea is already implemented in the TLM `peq_with_get` utility class of [1]. However because we model hardware FIFOs that are bounded, writing may be blocking too, so the Smart FIFO associates a time stamp with each free cells too, which is used to update the writer local date. The time stamps are also used to compute accurately non-Kahn accesses (`is_full`, `get_size`, etc).

Internally, the Smart FIFO contains as many cells as the hardware FIFO it models. Each cell is either free or busy, and in addition to the data, we store both the last data insertion date and the last freeing date for each cell. One index points to the first free cell and another to the first busy cell.

Here is how the `write` method works:

1) if all cells are busy, synchronize the writer process and wait until a cell is available (1 context switch)
2) if the first free cell freeing date is in the future, then increase the writer process local time up to this date
3) update the cell: fill the data and set the insertion date; advance the first free cell index
4) wake up a blocked reader process, if any.

The `read` method is symmetrical: wait until a cell is busy, next increase the reader process local time up to the insertion date of the first busy cell (if needed), update this cell state and the busy index, notify an event to the write side, and finally return the data.

B. The Smart FIFO: non-blocking interfaces

Modeling some hardware components, such as arbiters, cannot be done efficiently with decoupled threads because the synchronization points are too frequent. Another way to avoid the context switch cost is to use lightweight processes that always execute up to completion (SystemC SC_METHOD), and so have no context to store. Because SC_METHODs cannot use `wait`, the Smart FIFO must provide non-blocking interfaces. Obviously, we are only interested in cases where at least one side is accessed by a decoupled thread; otherwise, the user shall use a regular FIFO.

Typical read code for a SC_METHOD is:

```
if (fifo.is_empty()) {
  next_trigger(fifo.not_empty_event); return;}
x = fifo.read(); ...
```

Because the read is protected by the call to the `is_empty` method, the blocking method `read` can be reused as is, except that some code must be added to provide `not_empty_event` and `not_full_event`. Here *empty* and *full* refer to the external view of the FIFO, which differs of its internal state. Assuming the caller is synchronized, the method `is_empty` returns true if and only if: 1. either all cells are (internally) free, 2. or the insertion date of the first busy cell is the future. Thus, this method executes in constant time, but with two tests instead of one for a regular FIFO.

Notification of `not_empty_event` may occur in two places, corresponding to the two `is_empty` tests:

1) in the `write` method, when all cells were free
2) in the `read` method, when the next busy cell exists but has an insertion date in the future

In both cases, the notification is delayed until the insertion date of the new first busy cell.

C. The monitor interface

Contrary to regular FIFOs, getting the size of a Smart FIFO is not straightforward. For example, consider that a write is made at the global date $g = 10\,ns$ with a local date $l = 20\,ns$: the internal state of the Smart FIFO changes at $t = 10\,ns$ whereas the size of the real FIFO is incremented at $t = 20\,ns$. Thus, `get_size` is a function that must depend both on the internal Smart FIFO state and on the caller local date. Concretely, the `get_size` function must: 1. synchronize the caller, 2. iterate both over internal busy cells and internal free cells.

An internal **busy** cell is interpreted as a real busy cell if either:

- the insertion date is in the past
- **or** the previous freeing date is in the future (meaning that internally the cell has been freed and filled again since the `get_size` access date)

An internal **free** cell is interpreted as a real busy cell if both:

- the freeing date is in the future
- **and** the previous insertion date is in the past

Clearly, the Smart FIFO is slower than a regular FIFO for `get_size` accesses, but this function is rarely used in the applications we looked at (at most few accesses per seconds).

IV. Validation and Evaluation

A. Tests and Validation

For the validation of the Smart FIFO, we developed a test suite that covers its features. Most of the tests are deterministic but some are random. In those tests, the monitor interfaces are used extensively to follow how the FIFO sizes evolve.

Each test is executed in two modes: 1. using regular FIFOs and no temporal decoupling, 2. using the Smart FIFO and temporal decoupling; random tests use twice the same seed. Each test prints traces; each trace contains the local date of the process that printed it. Because the schedule is changed, the traces are not printed in the same order for the two modes: using temporal decoupling, dates may decrease when we switch from one process to the next. A test is considered as correct if after reordering of traces, both trace files are identical; meaning that the behavior and the timing are not changed at all.

Special care has to be taken about scheduler dependencies: indeed, it is well-known that because the SystemC scheduler is not fully deterministic, some pathological programs may have many valid behaviors depending on the scheduler choices [9]. Thus, we exclude such programs from the test suite.

To ensure that the test suite is complete enough, we have done mutation testing by hand. That is to say, we select a line in the Smart FIFO implementation, we modify something, we run the test suite again and check that at least one test fails.

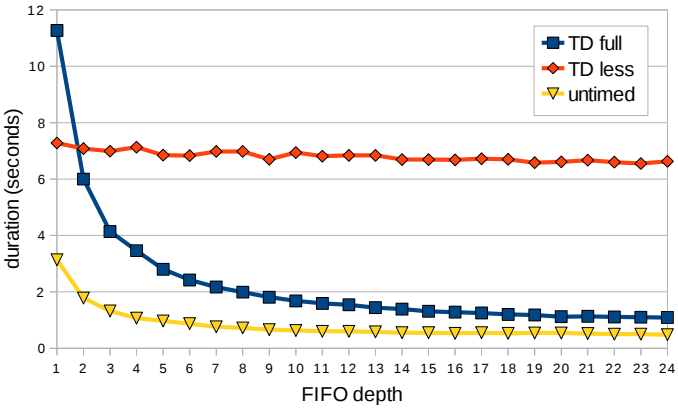

Fig. 5. Execution durations depending on the FIFO depth.

B. Performance Evaluation

The benchmark used for performance evaluation is a simple system with 3 modules (source, transmitter, and sink), connected by 2 FIFOs. 1000 blocks of 1000 words are transferred, with varying data rates. The FIFO depth is controlled by a parameter. We compare 3 implementations: 1. untimed with regular FIFO, 2. timed with no decoupling and regular FIFO (*TDless*), 3. timed with temporal decoupling and Smart FIFO (*TDfull*). Results are provided by Fig. 5.

The TDless model executes roughly at the same speed for all FIFO depths; there is always one context switch per access. On the contrary, untimed and TDfull models execute a context switch only when the FIFO is (internally) full or empty. Thus, larger is the FIFO, fewer are the context switches, and faster are the simulations. The TDfull model is always slower than the untimed model (about twice as slow), because more computations have to be done; that is the cost of timing accuracy. Compared to the TDless model that provides the same accuracy, our solution is slower for 1-cell FIFOs due to additional computations (same number of context switches), but is already faster for 2-cell FIFOs, and twice as fast for FIFO of depth 4. The gain factor reaches 6 for large FIFOs.

C. Case study: heterogeneous many-core SoC

The case study is an industrial heterogeneous many-core SoC for high performance computing [10]. Part of this SoC is composed of cores sharing a shared memory, but the most intensive computations, such as video decoding, are done by application-specific hardware accelerators, which communicates either directly with hardwired FIFOs or using a stream-based NoC. All communications done by TLM transactions (i.e., communication from cores to shared memory or to accelerators control registers) are temporally decoupled using existing methods. The remaining task was to apply temporal decoupling to the hardware accelerators.

Each hardware accelerator is modeled by a temporally decoupled thread. For each FIFO that connects directly two accelerators, replacing the regular FIFO by a Smart FIFO was straightforward. For the NoC itself, where a lot arbitration has to be done, we decided to model the routers using only non-decoupled SC_METHODs; thus NoC routers continue to use regular FIFOs. Accelerators and the NoC are connected through *network interfaces*. A network interface is in charge of packetizing data and arbitration among the incoming streams. Thanks to the possibility to use `inc()` in a SC_METHOD, we succeeded to model this module without any SC_THREAD. This module is connected to the accelerators using one FIFO per accelerator, and because accelerators are decoupled, we have to use a Smart FIFO here, which had to be slightly extended to manage efficiently the packetization.

After checking the new SoC model correctness using short tests, we evaluated the simulation speed using a longer benchmark test, which involves many hardware accelerators and one core for their control. The evaluation used an unmodified OSCI/ASI SystemC 2.2 kernel. We compared two versions: one using Smart FIFOs, and the other using FIFOs that calls `sync` at each access; both versions provide the same timing accuracy. The simulation duration changed from 38.0 to 21.9 seconds, giving a gain of 42.3% for this test.

V. Conclusion

Using the Smart FIFO that we introduced in this paper, one can add timing annotations in an untimed TLM model, without increasing the number of context switches. Compared to a timed TLM model using regular FIFOs with a context switch per access, we speed up simulations without any loss in timing or functional accuracy, and without requiring the user to set a time quantum.

The case study showed that the Smart FIFO is suitable for real-size industrial TLM models. The simulations are always faster, with a gain factor that depends on the relative cost of FIFO-based communications compared to other parts of the model. As of today, the final application developers continue to use the Smart FIFO.

References

[1] *IEEE 1666 Standard: SystemC Language Reference Manual*, Accellera Systems Initiative, 2011. [Online]. Available: http://www.accellera.org/

[2] L. Lehtonen, E. Salminen, and T. Ha andma andla andinen, "Analysis of modeling styles on network-on-chip simulation," in *NORCHIP 2010*, Nov. 2010, pp. 1–4.

[3] 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*, March 2006, pp. 94–99.

[4] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel simulation of SystemC TLM 2.0 compliant MPSoC on SMP workstations," in *DATE'10*, March 2010, pp. 606–609.

[5] R. Salimi Khaligh and M. Radetzki, "Efficient parallel transaction level simulation by exploiting temporal decoupling," in *Analysis, Architectures and Modelling of Embedded Systems*. Springer, 2009.

[6] M. Damm, C. Grimm, J. Haas, A. Herrholz, and W. Nebel, "Connecting SystemC-AMS models with OSCI TLM 2.0 models using temporal decoupling," in *Forum on Specification, Verification and Design Languages. FDL 2008.*, Sept. 2008, pp. 25–30.

[7] J. Cornet, F. Maraninchi, and L. Maillet-Contoz, "A method for the efficient development of timed and untimed transaction-level models of systems-on-chip," in *DATE'08*, March 2008.

[8] F. Herrera and E. Villar, "A framework for embedded system specification under different models of computation in systemc," in *DAC'06*. ACM, 2006, pp. 911–914.

[9] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy, "Automatic generation of schedulings for improving the test coverage of systems-on-a-chip," in *FMCAD'06*. IEEE, nov 2006, pp. 171–178.

[10] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *DATE'12*, March 2012, pp. 988–993.