Dedicated Compilation Techniques for SystemC using LLVM Compiler Infrastructure and JIT's Capabilities

Master student: *Si-Mohamed Lamraoui* Supervisors: *Matthieu Moy, Claire Maiza*
Verimag, 2, avenue de Vignate, 38610 GIERES - France
Email: <firstname>.<lastname>@imag.fr

***Abstract*—SystemC is a C++ library allowing the design of the hardware blocks contained in a System-on-chip at different level of abstraction. As SystemC is a C++ library, the programs may be compiled with a common C++ compiler. But these compilers miss a lot of optimization opportunities specific to SystemC programs. In this paper, we introduce a way to improve simulation performances of SystemC programs using the LLVM compiler infrastructure.**

***Index Terms*—SystemC, PinaVM, Tweto, TLM, LLVM, JIT, System-on-chip, compiler, optimizations**

I. Introduction

EMBEDDED systems are becoming increasingly complex over time. This is why the silicon industry needs new solutions and tools in order to stay on track.

An important point for the industries where products are outmoded quickly is the Time to market. It is the length of time it takes from a product being conceived until its being available for sale. To reduce the Time to market, the sofware and the hardware of these products must be design in parallel. Thus, in order to develop the sofware before the hardware is ready we model the System-on-chip of those embedded systems.

A System-on-chip is a single integrated circuit (chip) integrating all components of a computer or other electronic system. It contains a software to control the behaviour of the components. This sofware may be design without the hardware thanks to a model.

In this paper, we are interested in a high level of modelling, and more precisely in internal communications of the systems on chip. The standard *Transaction Level Modeling* (TLM) offers this level of abstraction and is popular in the industry. Although the models using TLM are faster than those using lower level ones, the need of better simulation performances is always on the agenda. Of course, the ultimate goal would be to simulate at the same speed as the real system.

To describe and simulate the Systems-on-chip we use SystemC. It is a C++ library allowing the design of the hardware blocks contained in a System-on-chip at different level of abstraction. The transactions between these blocks may be modelled using TLM.

As SystemC is a C++ library, the programs may be compiled with a common C++ compiler. However, these compilers miss a lot of optimization opportunities specific to SystemC programs.

In this paper, we introduce a way to improve simulation performances of SystemC programs. We reduce the number of operations induced by the TLM communications without corrupting the model.

We first introduce SystemC in Section 2. Then, in Section 3 we summarize related works that improve performance of simulation. We present the LLVM framework, the tool Tweto, the PinaVM compiler and the TLM Basic protocol in Section 4. In Section 5 we introduce our solution with concrete experimental results.

II. SystemC

SystemC [1], [2] is a C++ library aiming specifically at modeling Systems On Chip at differents levels of abstraction. As it is a C++ library, it uses the features of this language such as its syntax or its object oriented style. SystemC also inludes a simulation kernel to emulate the future system.

```
SC_MODULE(name) {
  ...
  // Constructor
  SC_CTOR(name) {
    ...
  }
};
```

Listing 1. A module

```
// input port of type porttype
sc_in<porttype>
// output port of type porttype
sc_out<porttype>
// inout port of type porttype
sc_inout<porttype>
```

Listing 2. Different kind of ports

A SystemC design is made of *modules* (see Listing 1), thereby the system is divided in less complex parts. A module allows designers to hide internal data representation and algorithms. Modules can only interact with each other through their interfaces. So, if a module is modified, the rest of the system has not to be modified while its interface is not changed. A module may contain serveral kinds of elements like nested module or processes.

Processes are the basic units of execution within SystemC. There are functions that are called to emulate the behavior of the target device. There are three types of processes: methods, threads and clocked threads. Unlike methods, threads can be suspended. In addition, the processes may use events that come from local or external signals to unlock suspended threads or to execute methods. The Listing 3 shows an exemple of process implementation.

Ports of a module are the external interfaces that pass information to and from a module, and trigger actions within the module. As shown in Listing 2, a port can be of three kinds. For instance, the input mode allows only data to come into the module from another one through a signal.

A signal creates connections between module ports allowing modules to communicate. Each signal is bound to two ports from two differents modules.

When the system is fully implemented, SystemC offers to simulate it. The SystemC simulations may be cycle-accurate, that means processes are executed and signals are updated at clock transitions. All of this is possible thanks to a build-in scheduler. Before the simulation starts, all modules are instantiated and properly connected, this is the *elaboration phase*.

```
SC_MODULE(name) {
  sc_in<type> in;
  void foo();
  SC_CTOR(name) {
    SC_METHOD(foo);   // or else SC_THREAD(foo);
    sensitive(in);
  }
};
void foo() {
  ...
}
```

Listing 3. Module with a method process

III. Related work

As far as we know, there is no previous work on automatic optimisation of TLM. However, there is a way to improve TLM transaction performances using the Direct Memory Interface (DMI) technique. This technique, included in the TLM-2.0 library, is done manually and should be done carfully [15].

An upward trend in the enhancement of simulation time is to make SystemC's process scheduler more effcients. Scoot [6], SplitPro [7] and the simulator SystemCASS [8] compute a fully static schedule before the beginning of the simulation.

On top of that, there are some commercial softwares. Unfortunately, these tools cannot be evaluated properly without their source code or research reports.

IV. Background

A. LLVM : Low Level Virtual Machine

Low Level Virtual Machine (LLVM) [9], [10] is a compiler framework that optimizes compilation, link, execution and idle time in programs written in any language. LLVM can be used to create code generators and optimizers for an arbitrary architecture.

This framework has been built around a dedicated code representation (bitcode). This intermediate representation is an abstract RISC-like instruction set based on a language-independent type-system. It is composed of high-level instructions while being low-level enough to represent any program. LLVM's code representation has many convenient features like its SSA (Static Single Assignement) [11] form that facilitates compilation optimization and analysis.

LLVM is a virtual machine, it has run-time capabilities that operate on the program during the execution giving several opportunities to improve performances. This last point is possible thanks to the Just-In-Time (JIT) technique [12], [13].

B. Tweto

Tweto is based on simulation with elaboration-time optimizations. Classic compilers, such as `GCC`, do not take into acount all the data that become constant after the elaboration phase. Thus, these compilers miss a lot of optimization opportunities. Tweto exploits these data to do process specialization, indirect call resolution and calls specialization.
Claude Helmstetter is the main contributor of this tool.

C. PinaVM

PinaVM (PinaVM is not a Virtual Machine) [14] is a SystemC frontend built using the LLVM framework. It was introduced to perform efficient formal and symbolic verification of SystemC programs. Basically, PinaVM takes an LLVM bitcode as input, this bitcode represents the SystemC program. Then, using the LLVM's JIT engine, PinaVM executes the elaboration phase. At this point, all modules are created and connected. We know this configuration will remain static over time. Then, PinaVM retrieves all the information about the system's architecture. Different backends use these informations to carry out verifications.
In this paper, we add to PinaVM a backend aimed toward simulation and optimizations. We use the tool Tweto as a basis to make this backend. We also extend it by including TLM optimizations.

D. TLM Basic Protocol

Transaction-Level Modeling (TLM) is a high-level approach to model digital systems where the details of the communication among modules are separated from the details of the implementation of functional units or of the communication architecture [15], [16], [17]. TLM allows to create different kinds of protocols, like the *Basic Protocol*.

The Basic protocol is not used for production, it was introduced as an example. It is a simple *Bus* protocol with two transaction modes: Read and Write. Each transaction is routed according to an address map through the *Bus*. Figure 1 shows an exemple of system that uses this protocol. A typical transaction starts from an *Initiator module* (CPU) that initiates the communication. The data is then sent to the *Bus*. When the given address aims an existing *Target module* (RAM), the data is redirected to this target by the *Bus*.

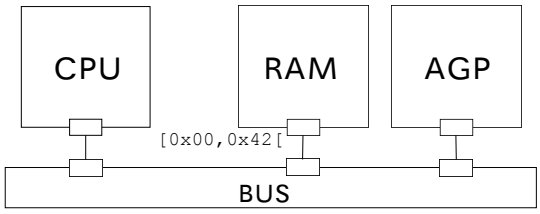

Fig. 1. A system using TLM Basic protocol. The RAM (target module) is reachable in the range of addresses 0x00 to 0x42.

In Listing 4 the implementation of the system shown in Figure 1. Here, we just show a simplified code of the CPU and RAM module. The CPU module writes on the Bus through a socket in the process named *thread*. The RAM module implements the write method, witch saves the received data in an array. In the beginning of the main function the modules are instantiated. Then, the Bus mapping is set for each target modules. Next, the modules are connected to the Bus. Finally, we start the simulation.

V. Static Address Resolution

The TLM Basic protocol is slower than direct access even if it is a straightforward protocol. A simple write transaction uses a lot of operations. In Figure 2, a CPU module attempts to write the data `data` at the address `addr`. The transaction starts by a virtual call on the socket (1). Then, the call is forwarded to the target socket (2). The bus decodes the address (3) and does another virtual call (4). The call is forwarded again to the target. Finally, the actual method is called in the RAM module (5).
Knowing this, we aim at improving the way the transactions are done to get better performances.

A. Our solution

To avoid all the intermediate operations, we choose to replace the original call by a direct call to the target module's method.

For implementing this solution, we created an optimisation pass in the Tweto backend. As shown in Figure 3, the pass is run after the elaboration phase and before the simulation.

```
SC_MODULE(cpu) {
  basic::initiator_socket<initiator> socket;
  void thread(void) {
    socket.write(addr, data);
  }
  SC_CTOR(cpu) {
    SC_THREAD(thread);
  }
};

SC_MODULE(ram) {
  basic::target_socket<target> socket;
  status write(addr_t &a, data_t &d) {
    mem[a] = d;
    return tlm::TLM_OK_RESPONSE;
  }
};
...
int sc_main (int argc, char **argv) {

  Cpu intel_cpu("Intel-8088");
  Ram simm_ram("SIMM30");
  Agp onex_agp("AGP1X");
  Bus isa_bus("ISA");

  // SIMM30 is mapped at addresses [0, 42[
  isa_bus.map(simm_ram.socket, 0, 42);
  // AGP1X is mapped at addresses [42, 84[
  isa_bus.map(onex_agp.socket, 42, 84);

  // connect components to the bus
  intel_cpu.socket.bind(isa_bus.target);
  isa_bus.initiator.bind(simm_ram.socket);
  isa_bus.initiator.bind(onex_agp.socket);

  // and start simulation
  sc_start();

  return 0;
}
```

Listing 4. Implementation of the system shown in Figure 1

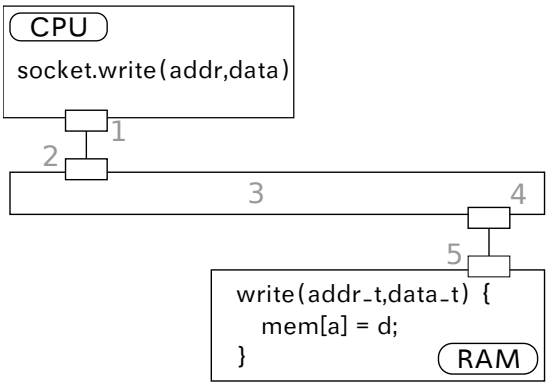

Fig. 2. A write transaction. 1. Virtual call on the socket, 2. Forwarded to the target socket 3. Address decoding 4. Forwarded to the target socket 5. Call of the actual write method

Since the elaboration phase is done at an earlier level, the pass know how the modules are connected. For instance, in Figure 2, it will determine that the called method belongs to the RAM module.

As we bypass the *Bus*, the automatic redirection of the transactions will not be done anymore. Thus, before replacing a call, we need the address used to find out which module is targeted.

To do so, we use a frontend's method designed to retrieve a value from the bitcode. This method uses the JIT engine in order to execute

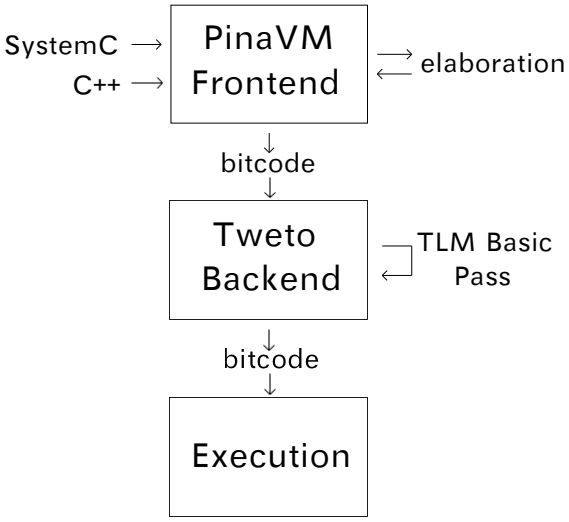

Fig. 3. Big picture on simulation of SystemC programs.

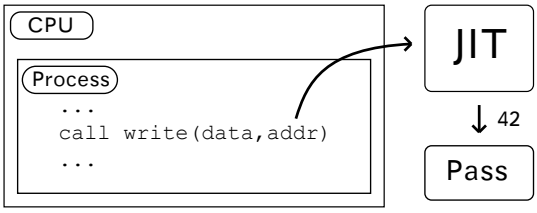

Fig. 4. Address value retrieving with the JIT engine..

the appropriate piece of code on the fly. As shown in Figure 4, the call to the write or read method is searched in the CPU module's proccess. Then, the frontend's method is run on the address argument. If the value has been extracted, the pass can make the appropriate steps.

At this stage, we cannot make any modifications in the parent processes of those calls. As the bitcode of the processes' methods are shared by the same kind of modules, we cannot change them. When there is a change to do, we first clone the original process.

When the initiator process is cloned, we can search the write or read methods in the targets modules. As LLVM allows to look for a method by name, we are able to find them using the *name mangling*. Originally, the name mangling is a technique that creates a unique name in the bitcode for each methods. Here, we exploit this technique because we know how the unique name of the write and read methods are formed, as shown in Figure 5. All mangled symbols begin with `_Z`, this is followed by `N`. Then a series of `<length, id>` pairs (the length being the length of the next identifier), and finally `E`, followed by a code that represents the type of the method (prototype). In our case, there are only two pairs : `<len,module type>` and `<5,write>` or `<4,read>`.

(1) `_ZN + moduleType + 5write + E + ERKjS1_`
(2) `_ZN + moduleType + 4read + E + RKjRj`
(3) `_ZN6target5writeERKjRj`

Fig. 5. 1,2. Write and read method's mangled symbol decomposition. 3. An exemple of mangled symbol.

Once the method is retrieved, we replace the indirect calls in the CPU module by direct ones. Figure 6 illustrates the code modification in the cloned CPU's process after applying our solution. Now, all the costly operations are avoided and there will just be only one direct call.

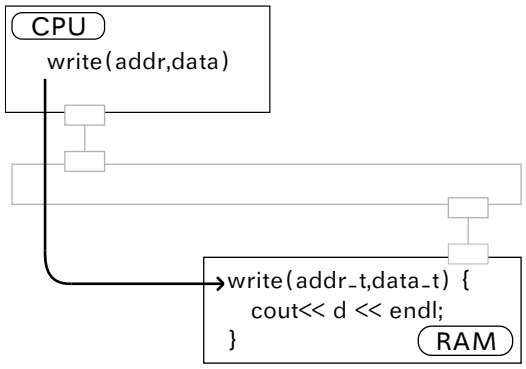

Fig. 6. Direct call to the RAM's method from the CPU module.

Before returning control to the SystemC program, we ought to recompile and relink every modified/created methods. This is done by the LLVM's execution engine.

B. Experiments

In this section we show the first results we obtained. The experiment consists of running a lot of transaction between several initiator modules and one target module. Each initiator makes 10000 writing to the target. The number of initiators for each measure increases from 10 to 100.

Figure 7 shows the simulation time depending on the number of initiators. Although the use of our pass adds computation time, the simulation time is already lower from 10 initiators. We gain at most $200ms$ with 100 initiators that shows the efficiency of our solution.

The improvement due to our pass is huge on this toy exemple and should be non-negligible on real programs.

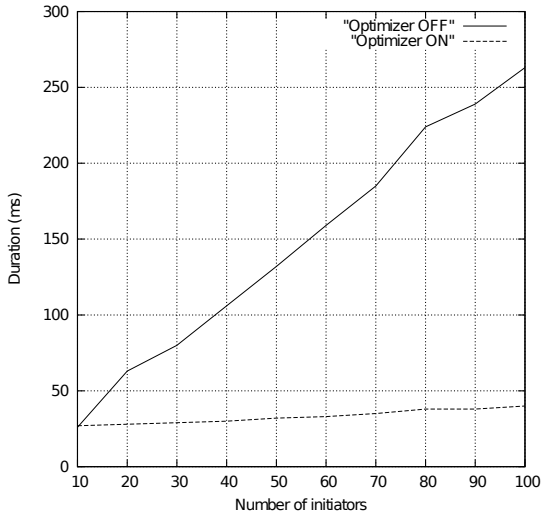

Fig. 7. Initiators vs. one target experiment (10.000 calls).

VI. Conclusion and Future Work

In this paper we have presented an extension of PinaVM, a tool based upon the framework LLVM, which facilitates the developement of compilers.

Initially PinaVM was introduced to perform efficient formal and symbolic verification of SystemC programs. We extended it to allow simulation of SystemC programs thanks to the previous work of Claude Helmstetter. Moreover, we included an optimization pass that reduces the number of operations induced by the transactions of the TLM Basic Bus protocol. The primary results of these optimizations were very encouraging. We shown that it is possible to get huge gains on simple exemples.

Unfortunately, due to the short time spent on this work, our pass does not handle enough particular cases to be used in bigger systems. Thus, much more work is necessary in order to enhance this optimization pass.

VII. Acknowledgement

The author would like to thank Kevin Marquet and Claude Helmstetter for the assistance they have provided on PinaVM and Tweto, Claire Maiza and Matthieu Moy for their precious help and support during this work.

References

[1] "The Open SystemC Initiative." [Online]. Available: http://www.systemc.org

[2] "SystemC 2.0 user's guide - Update for SystemC 2.0.1," 2002. [Online]. Available: http://www.systemc.org

[3] H. Alemzadeh, S. Aminzadeh, R. Saberi, and Z. Navabi, "Code optimization for enhancing systemc simulation time," in *Design Test Symposium (EWDTS), 2010 East-West*, sept. 2010, pp. 431 –434.

[4] J. Ditmar, "Area optimisation in systemc hardware compilation," 2007.

[5] S. A. Homa Alemzadeh, "Efficient system level design by optimizing systemc simulation time," Faculty of Electrical and Computer Eng., School of Engineering, University of Tehran.

[6] N. Blanc, D. Kroening, and N. Sharygina, "Scoot : A tool for the analysis of systemc models," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer Berlin / Heidelberg, 2008, pp. 467–470, 10.1007/978-3-540-78800-3_36. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3_36

[7] Y. N. Naguib and R. S. Guindi, "Speeding up systemc simulation through process splitting," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '07. San Jose, CA, USA: EDA Consortium, 2007, pp. 111–116. [Online]. Available: http://portal.acm.org/citation.cfm?id=1266366.1266392

[8] A. Buchmann, R.; Greiner, "A fully static scheduling approach for fast cycle accurate systemc simulation of mpsocs," in *Microelectronics, 2007. ICM 2007. Internatonal Conference on*, LIP6/UPMC, Univ. Pierre et Marie Curie, Paris, 2007, pp. 101 – 104.

[9] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[10] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002, *See* `http://llvm.cs.uiuc.edu`.

[11] C. McConnell and R. E. Johnson, "Using static single assignment form in a code optimizer," *ACM Lett. Program. Lang. Syst.*, vol. 1, pp. 152–160, June 1992. [Online]. Available: http://doi.acm.org/10.1145/151333.151368

[12] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, pp. 97–113, June 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857077

[13] M. P. Plezbert and R. K. Cytron, "Does 'just in time' = 'better late than never'?" in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '97. New York, NY, USA: ACM, 1997, pp. 120–131. [Online]. Available: http://doi.acm.org/10.1145/263699.263713

[14] K. Marquet and M. Moy, "PinaVM: a SystemC front-end based on an executable intermediate representation," in *International Conference on Embedded Software International Conference on Embedded Software*, Scottsdale, USA, 10 2010, p. 79, SD B.4.4, I.6.4, D.2.4 OpenTLM (projet Minalogic). [Online]. Available: http://www-verimag.imag.fr/ moy/publications/pinavm-emsoft.pdf

[15] "Osci tlm-2.0 language reference manual," 2009. [Online]. Available: www.systemc.org

[16] J. P. Adam Rose, Stuart Swan, "Transaction level modeling in systemc."

[17] S. Pasricha, "Transaction level modeling of soc with systemc 2.0."