Simulation-based Equivalence Checking between SystemC Models at Different Levels of Abstraction *

Daniel Große[1] Markus Groß[1] Ulrich Kühne[2] Rolf Drechsler[1]

[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{grosse,mgross,drechsle}@informatik.uni-bremen.de
[2]LSV ENS de Cachan, 94235 Cachan, France
kuehne@lsv.ens-cachan.fr

ABSTRACT

Today for System-on-Chips (SoCs) companies Electronic System Level (ESL) design is the established approach. Abstraction and standardized communication interfaces based on SystemC Transaction Level Modeling (TLM) have become the core component for ESL design. The abstract models in ESL flows are stepwise refined down to hardware. In this context verification is the major bottleneck: After each refinement step the resulting model is simulated again with the same testbench. The simulation results have to be compared to the previous results to check the functional equivalence of both models. For models at lower levels of abstraction strong approaches exist to formally prove equivalence. However, this is not possible here due to the TLM abstraction. Hence, in practice equivalence checking in ESL flows is based on simulation. Since implementing the necessary verification environment requires a huge effort, we propose an equivalence checking framework in this paper. Our framework allows to easily compare variable accesses in different SystemC models. Therefore, the two models are co-simulated using a client-server architecture. In combination with multi-threading our approach is very efficient as shown by the experiments. In addition, the time required for debugging is reduced by the framework since the respective source code references where the variable accesses did not match are presented to the user.

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: [Computer-Aided Design (CAD)]

General Terms

Verification

Keywords

Transaction Level Modeling, Equivalence Checking, Debugging, SystemC

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 01M3088.

GLSVLSI'11, May 2–4, 2011, Lausanne, Switzerland.

1. INTRODUCTION

For a long time *Register Transfer Level* (RTL) has been sufficient for modeling. But around 2000 *System-on-Chips* (SoCs) reached a complexity which made higher abstraction mandatory. This strategy was implemented and *Electronic System Level* (ESL) design emerged [2]. A widely accepted language for ESL design is SystemC [20, 14, 15, 13]. As a C++ class library SystemC perfectly supports abstract modeling, in particular *Transaction Level Modeling* (TLM) [6, 11].

In an ESL design flow starting from the textual specification, the algorithmic design is specified first. At this level the basic functionality of the system is explored and defined no matter which parts become hardware or software. Afterwards the system is refined to a TLM model. This model consists of modules communicating over channels, i.e. data is transferred in terms of transactions. Within TLM different levels of timing accuracy are available such as untimed, loosely-timed, approximately-timed and cycle-accurate. The respective levels allow e.g. for early software development, performance evaluation and HW/SW partitioning. As the last step of the ESL flow the hardware part of the TLM model is refined to RTL.

However, the dominating factor in ESL design is still verification. Following the refinement steps the same testbench can be basically reused at lower levels applying the transactor concept of SystemC. By this, during simulation the same high-level input is fed into both models but a huge effort is required to implement the verification environment which has to perform the actual equivalence check.

In the pure hardware domain where lower level models are considered – e.g. RTL or gate-level models – *Equivalence Checking* (EC) based on formal methods has been very successful [12, 8]. A distinction is made between combinational EC and sequential EC. In combinational EC two designs without memory elements are tested for equivalence. In contrast, sequential EC aims to verify whether two sequential circuits have the same functionality. If the state encoding is the same, the problem can be reduced to the combinational case. Otherwise, the problem is much harder, but a lot of progress has been made (for an overview see e.g. [19]). However, all these methods can only be applied if both models can be compiled into a circuit-like representation which is not the case for TLM descriptions.

Recently, there has been research targeting equivalence checking between system-level models and RTL. Basically, the idea of [23] is to identify sequential compare points, i.e. variables which should have the same value with respect to the considered time domain. But this approach requires explicit timing information which is not available at TLM. The authors of [5] propose the notion of event-based equivalence for SystemC TLM and RTL models. The

theory of [5] has not been implemented for this scenario (the experiments presented consider the setting of abstracting an RTL model to TLM) and needs manual interaction to relate the events at different abstraction levels. Other works such as e.g. [18, 7] relate C programs to hardware description languages. But these approaches all assume strict timing information to be available. The authors of [10] present first steps for equivalence checking using formal methods in combination with SystemC but do not consider TLM models. The specific problem of handling memories in formal equivalence checking of system-level models against RTL has also been considered [17].

In this paper we present a framework for equivalence checking between SystemC models described at different abstraction levels. In particular, we support TLM designs as typically used in ESL flows. The framework gives a pragmatic and easy-to-use solution for comparing model behaviors. Hence, a lot of time is saved which usually has to be spent for implementing the respective verification environment. Based on minor extensions of the models the developed client-server architecture allows to simulate and compare the results very efficiently. The verification engineer chooses which variables are used for comparison, e.g. internal variables, variables of the interfaces, members of transactions or even arbitrary C++ objects. In addition, the verification engineer specifies whether a different ordering of accesses to a variable is valid. For instance, a CPU refined to a lower abstraction level may implement out of order execution and hence some data values can be different at certain time points. In general, a trade-off between performance and accuracy is possible based on the number of variables included in the comparison.

Then for equivalence checking, both SystemC models (the clients) are co-simulated (controlled by the server) and thereby the comparisons are carried out.

Different methods for co-simulation using SystemC have been proposed in the literature, see e.g. [3, 9, 22]. However, all these papers consider SystemC-based environments for co-simulation of hardware and software, i.e. connecting different worlds. To the best of our knowledge no framework has been proposed supporting the design and verification teams in equivalence checking when refining the system to lower levels of abstraction. We close this gap here and improve the productivity of SystemC-based ESL design flows.

Overall, we summarize the contributions of this paper as follows:

- Equivalence checking framework
  A framework for equivalence checking of SystemC models in form of a library is introduced. The framework provides easy-to-use interfaces to compare the behavior of different SystemC models by only minor extensions of the models.
- Efficiency
  The client-server architecture of our framework makes intensive use of multi-threading. Thereby, the approach benefits from multi-core systems and as a result very fast variable comparisons can be performed.
- Debugging
  The generated equivalence report summarizes important information for the user. In particular, in case of non-equivalent behavior of the checked SystemC models the developed approach helps to detect and locate the bug quickly due to reported variable access information.

We present experimental results demonstrating the advantages of our framework. For a CISC CPU developed in a top-down ESL design flow two major bugs have been found using this framework.

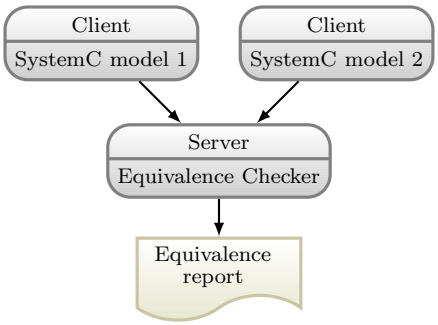

Figure 1: Overall Flow

2. EQUIVALENCE CHECKING OF SYSTEMC MODELS

In this section the equivalence checking approach to compare the behavior of different SystemC models is introduced. For this task we have developed a framework which has been built as a client-server architecture. The equivalence checking server interacts with exactly two SystemC models, which in turn act as clients. Three phases need to be followed using the proposed approach: model extension, equivalence checking, and report analysis. Before we explain these phases in detail, the overall flow is presented.

2.1 Overall Flow

Figure 1 depicts the overall flow of the proposed approach. At the starting point there are two SystemC models which are to be compared. They have to be extended in order to function as clients for the equivalence checking server. This extension process consists of two steps. First, a model has to be analyzed to determine which variables[1] should be used for comparison during equivalence checking. Second, the verification engineer extends the model by using our library. In doing so, the SystemC model is augmented with networking functionality and can connect and communicate as client with the EC server. The extended model is then compiled into an executable. This process needs to be performed for all SystemC models whose equivalence is to be checked. Since only minor extensions of the SystemC models have to be carried out this is a very convenient process. Furthermore, this procedure fits perfectly into the refinement steps of typical ESL design flows.

To perform the equivalence check our server first has to be executed. Then, both models have to be started to begin their simulation. During the equivalence check both clients send their data to the server. The server collects the data and continuously checks if the two models show the same behavior. Finally, at the end of the equivalence check a report is generated. This report summarizes the results of the comparison for each variable over the simulation time, e.g. if there have been mismatches during read or write of a concrete variable as well as the frequency of such events.

In the following the three phases are described in more detail.

2.2 Model Extension

2.2.1 Analysis

Each model contains a set of variables important to the model's behavior. The verification engineer determines which

[1]We use the general term of variables here to cover arbitrary C++ objects which of course also include specific SystemC types. Concrete examples are presented in the following sections.

variables should be compared in the equivalence checking phase. A huge list of variables covered by the check is likely to decrease the performance. The more variables are chosen, the more data has to be sent to the server and stored in memory (for some time). Therefore, a minimal but sufficient set of variables should be selected.

In addition, the verification engineer needs to decide whether a read and/or write action on a variable is crucial to the model's behavior. If the value read from a variable is not needed, it should not be sent to the server. This way the data transferred through the communication sockets is minimized.

2.2.2 Extension

The next step is the extension of the SystemC model. The global variable `ec` is declared at first. This variable contains an instance of the developed class `EquivCheck`. This class is part of our equivalence checking framework and comes with our library `libec`. The class `EquivCheck` provides the functionality to communicate with the equivalence checking server as well as functions to help recording the required data. Moreover, it offers methods to register a set of variables to include them in the equivalence check.

The verification engineer selects a list of variables that shall be used to determine whether the two models behave in the same way. The next step is to spot all locations in the model's source-code where one of these variables is accessed in a reading and/or writing way. Once these locations have been determined, every access needs to be reported to the server. The data sent to the server contains the following information:

1. the id of the variable (specified at registration),
2. the access type (read or write),
3. the address where the variable is accessed (particularly important for arrays),
4. the current value in case of a read access, the new value in case of a write access.

In the following we give a concrete example demonstrating the model extension.

EXAMPLE 1. *Consider a CPU and a memory while the memory is a separate module in a SystemC TLM model. Assume, the CPU wants to read the value of a specific memory address. The CPU communicates with the memory using the blocking transport interface of TLM-2.0 [21]. So in the TLM memory the method* `b_transport` *as shown in Figure 2 is implemented. The memory itself is modeled as an integer-array inside the memory module. In line 10 the value of the memory address (given by* `addr`*, which has been extracted from the payload; see line 4) is read from the array and thereafter copied to the transaction. In this example the verification engineer decided to compare also read accesses with the second SystemC model and hence wants to report these to the equivalence checking server. This is done from line 13 – line 14. The function* `report` *is available via our* `EquivCheck` *class. It takes the id of the variable (*`ID_MEM`*), the type of access (*`ACCESS_READ`*), the value of the memory address as string and the address of the access as parameters. As a result of calling this function the data is processed and sent to the server.*

Figure 3 depicts the same function implemented in the RTL model of the memory. Again, the value read from the memory is sent to the server (see line 12 – line 13).

```
1  void Memory::b_transport(tlm::tlm_generic_payload& trans,
2                           sc_time& delay) {
3    tlm::tlm_command cmd = trans.get_command();
4    sc_dt::uint64 addr = trans.get_address();
5    size_t length = trans.get_data_length();
6    uint32_t* data = reinterpret_cast<uint32_t*>
7                     (trans.get_data_ptr());
8
9    if (cmd == tlm::TLM_READ_COMMAND) {
10     memcpy(data, &mem[addr], length);
11
12     // report value read from memory
13     ec.report(ID_MEM, ACCESS_READ,
14               toString(*data), addr);
15
16   } else if (cmd == tlm::TLM_WRITE_COMMAND)
17     [...]
18
19   trans.set_response_status(tlm::TLM_OK_RESPONSE);
20 }
```

Figure 2: TLM model: read memory value

```
1  void Memory::entry() {
2    if (req.read()){
3      ack = true;
4      if (rw.read()) // write access
5        [...]
6      else { // read access
7        int addr = addr_i.read();
8        data_o = (mem[addr], mem[addr + 1],
9                  mem[addr + 2], mem[addr + 3]);
10
11       // report value read from memory
12       ec.report(ID_MEM, ACCESS_READ,
13                 toString(data_o.read()), addr);
14     }
15   }
16 }
```

Figure 3: RTL model: read memory value

As can be seen in the example based on our library only a single function call to use the respective data for equivalence checking has to be inserted into a SystemC model.

In the following the equivalence checking phase of the proposed approach is described.

2.3 Equivalence Check

In the equivalence checking phase the server collects data from the two clients and analyzes this data. On top of the client-server architecture the server functionality has been separated into multiple threads. For each connected client one thread collects the incoming data. Furthermore, for each registered variable a thread is created to compare the respective data. This extensive use of threads greatly improves the overall performance on multi-core systems as shown later in the experiments. Before we describe the equivalence checking process, we define under which conditions two models are considered to show the same behavior.

Given two SystemC models, M_1 and M_2, let the variable set of M_1 be V_1 and of M_2 be V_2. Moreover, based on the model extension phase let the set of selected variables be $S_1 \subseteq V_1$ and $S_2 \subseteq V_2$, respectively.

As we consider the model behavior of both models with respect to simulation there is a certain timestamp t describing the current progress of the overall simulation. Now, the set S_1^t (S_2^t) describes the values of all selected variables of model M_1 (M_2) at time t.

Since all models send their data in a normalized form to the EC server, each model requires a transformation function to accomplish this normalization. More precisely, the function f_i normalizes the values of a set of selected variables

to allow the equivalence checking server to compare them. For example, if in a high level model configuration variables have been declared separately, and in the corresponding RTL implementation they have been embedded into different configuration registers, the transformation function extracts the separated values and combines them to a single one[2]. Now, we can define the equivalence criterion:

DEFINITION 1. *Two models M_1 and M_2 are equivalent with respect to the selected variable sets S_1 and S_2 at time t, iff*

$$f_1(S_1^t) = f_2(S_2^t).$$

For this definition we have to specify when two normalized values of a variable match: Due to the event-based scheduler of SystemC it may happen that a SystemC process is executed more than once in the current simulation cycle. In this case the process may report intermediate values to the EC server, which are irrelevant. But this can be easily recognized by keeping the last sent value in the server, i.e. all values up to this last sent value have been computed in previous delta cycles and will be discarded[3]. Finally, the following definition results:

DEFINITION 2. *During equivalence checking of two SystemC models two variables at a certain simulation cycle match, iff both latest (stabilized) values with respect to the delta cycle concept are equal.*

Using these definitions the equivalence checking works as follows: After compilation of the extended models the server has to be executed. Then, both models have to be started. In the beginning of their simulation each model connects to the server. As a second step they register all variables they want to compare during the equivalence check at the server. Based on the registration id variables are matched. Variables only present in one of the two models are discarded with a warning. This process takes place before any of the considered variables are read or written.

Next, the simulation of the models continues. While their simulation advances, registered variables in the SystemC models are read and written. Since the models have been extended, these accesses are sent to the server and are stored in the corresponding data structures. The main data structures used in the server are based on *Standard Template Library* (STL) deques [1]. This type offers performance enhancements compared to lists or vectors because deques grow much more efficiently and avoid reallocation of its elements. Also the equivalence checking server has to handle large amounts of data and the deques have to grow and shrink very frequently. While both models are running and sending data to the server, the server itself collects the data and tries to match them according to the definitions from above. Typically, we use a fixed block size (standard value is 100,000) for storing incoming accesses. If this number is exceeded in both models, a process matching the variables is invoked. To reduce the memory usage of the equivalence check every successful compared pair of values is removed immediately, but added in the statistics. We explain the respective details in the next section.

2.4 Report Analysis

Finally, after both models have finished their simulation, the server generates a report. This report contains important statistical information for the verification engineer. It lists all variables divided into read access and write access. In addition, several counters are associated to each variable, displaying the number of *total accesses*, *matched accesses*, *unmatched accesses* and *swapped accesses*. Total accesses equal the number of all accesses, and matched accesses follow Definition 2. An unmatched access is an access that appeared in only one model and could not be matched to an access in the other model. These accesses suggest a difference between the behavior of both models and need further investigation on behalf of the verification engineer. Swapped accesses may occur when a model performs the same accesses to a variable but in a different order. This may happen for instance in a CPU implementing out-of-order execution. In this case the instructions that are executed may be dynamically reordered by the CPU and therefore cause a slightly different order of the variable accesses. The user specifies a maximal window size w which is used to match the second value by considering all values up to w steps in the future of the other SystemC model. Extending Definition 1, this behavior is formally described using an out-of-order function o which permutes a set of variable sets starting from a specific time t up to time $t + w$. This yields the following definition of equivalence:

DEFINITION 3. *Two models M_1 and M_2 are equivalent with respect to the selected variable sets S_1 and S_2 and the out-of-order function o at time t, iff*

$$f_1(S_1^t) = f_2(o(S_2^t, S_2^{t+1}, \ldots, S_2^{t+w})).$$

Finally, due to the reported statistical information the verification engineer is able to determine which variable has been read and/or written differently and the frequency of such an event. It is also shown if only read or write accesses are unmatched or both. This statistical information has to be analyzed by the verification engineer.

In the following we present an example of an equivalence report.

EXAMPLE 2. *Consider again the CPU and the memory presented in Example 1. We have shown already how reading a value from a specific memory address has been implemented (and sent) in a TLM model (see Figure 2) and in the corresponding RTL model (see Figure 3), respectively. Assume, both models are checked for equivalence. It is examined whether the same read accesses to the memory give equivalent results (see also model extension phase in the last section). The resulting report is shown in Table 1. As can be seen, there are many unmatched read memory accesses. This means that both models behave differently. To track down the error the code for reading from a memory address has to be analyzed in both models (from there the variable accesses have been reported). From Table 1 we see that there are matched cases as well. Hence, we compared only a few simulation scenarios for both results, i.e. where (a) the values for* `data_o` *matched and (b) the values for* `data_o` *that did not match. In conclusion, the bug was implemented in the RTL model while refining the TLM model (see line 8 – line 9 in Figure 3): Four 8-bit values are assigned to the variable* `data_o`*, but they have been arranged in the opposite order. Figure 4 gives the correct assignment to* `data_o`*. Using our equivalence checking approach we were able to show that both models have the same behavior after the fix.*

In the next section the proposed approach is studied for a CISC CPU.

3. EXPERIMENTAL EVALUATION

In this section the proposed approach is exemplified for a CISC CPU used for educational purposes. First, the basics of the CPU are briefly described. Following a top-down

[2]Often the transformation function is nothing else than the identity.

[3]We use `sc_time_stamp` to identify the current simulation cycle, thereby revealing the timestamp advance.

Table 1: TLM and RTL equivalence report

Equivalence check of TLM and RTL					
Reading variable accesses					
	ID	Total	Matched	Swaps	Unmatched
MEMORY	0	1,452,524	85,532	0	**1,366,992**

```
1    [...]
2  else { // read access
3    int addr = addr_i.read();
4    data_o = (mem[addr + 3], mem[addr + 2],
5              mem[addr + 1], mem[addr]);
6    [...]
```

Figure 4: RTL model: read memory value (corrected)

ESL design flow, the hardware implementation of the CPU has been developed. During the respective refinement steps we applied the proposed equivalence checking approach to verify the consistency of the SystemC models. The different phases to apply our approach are demonstrated and the results are discussed. Finally, we also evaluate the performance gains resulting from the client-server architecture using multi-threading.

3.1 CISC CPU

We consider a modified version of the Y86 CPU [4]. The Y86 CPU is a simple CISC CPU implementing a subset of the instructions of the IA-32 architecture [16]. It has been designed as von Neumann architecture. The data width of the memory is 8 bit. In the modified version the memory size is 8 KByte. The CPU has 8 registers with 32 bit data width each. In total, the Y86 has 8 instructions: 2 arithmetic instructions, 3 load/store instructions, 2 jump instructions and 1 other instruction.

For the Y86 an *Instruction Set Architecture* (ISA) model is available [4], consisting of 289 lines of code in a slightly modified version. This model, which is written purely in C++, served as our high-level starting point for the ESL design flow. We refined this model to a SystemC TLM model (465 lines of code) using TLM features like e.g. socket communication to connect the memory and the registers to the CPU. Next, the TLM model has been refined again to SystemC RTL (661 lines of code). The final RTL model implements a 5 stage pipeline, which is not part of the ISA model.

In the following the process of extending a model is described for the Y86 CPU at hand.

3.2 Y86 Model Extension

The different models of the Y86 CPU have several variables in common. To perform the equivalence check a set of variables had to be determined, which is used during the process to check the correspondence of both models. All models have the register and the memory variables in common, i.e. during refinement additional details have been added (e.g. pipeline stages etc.). But from the functional perspective the basic CPU registers as well as the memory remained unchanged. Moreover, they describe the state of the CPU at a fixed point in time independent of the level of abstraction. Therefore, we selected these two variable arrays as variable sets S_i for the equivalence check. To complete the extension process we augmented the models using the `libec`. Hence, the models report the mentioned variable accesses.

3.3 Y86 Equivalence Checking

For the initial ISA model several programs for validation have been developed. They have been successfully executed on the ISA model. After each refinement step we also executed the programs on lower level models, i.e. TLM and RTL, respectively. However, without the proposed framework a direct and precise comparison requires a significant effort for implementing the necessary validation environment. In contrast, with the framework we can easily and efficiently check whether the models show the same behavior.

```
1   [...] = MEM[ea] | (MEM[ea+1] >> 8) |
2           (MEM[ea+2] >> 16) | (MEM[ea+3] >> 24);
```

Figure 5: ISA model: read memory value

3.3.1 ISA and TLM model

First, we checked the equivalence of the Y86 ISA model and the Y86 TLM model. As it turned out, they gave the same results for most of the input programs, but failed e.g. for a program implementing the bubble-sort algorithm. The equivalence report revealed similar numbers as shown in Table 1, i.e. most of the memory accesses did not match. The input programs with just a few memory accesses did not produce any unmatched memory accesses.

To find the source of the problem we compared the actual values of a read and write access to the memory. The first few accesses were correct, but later almost no access matched. As the values read or written to the memory are 32 bit wide, but the memory has an 8 bit data width, the values are divided into four 8 bit parts. A value pair resulting from an unmatched memory read looks for example like this (least significant byte is left):

```
ISA: f3-00-00-00 TLM: f3-d4-aa-b5
```

Each time the first 8 bit of both values were equal and the remaining 24 bits were zero in the ISA model. The correct value is the one of the TLM model because it was initially written into the memory as test-data for the bubble-sort algorithm. This points towards an error in the ISA model. After comparing the memory access functions in both models the error was found in the read memory function in the ISA model. The relevant part of the function is depicted in Figure 5. The variable `ea` denotes an offset for the memory access and is irrelevant here. The ISA model divides an 32 bit value into four 8 bit blocks using right shifts when writing to the memory. These four values have to be shifted back when being read from the memory. But instead of shifting to the left, the ISA model again shifted to the right. This way 3 of the 4 bytes became zero. The byte which did not need to be shifted is read correctly from the memory.

This bug caused that all the programs which read only small values (not larger than `FF`) passed the equivalence check successfully. To correct this bug we simply had to change the right shifts into left shifts. Overall, with the proposed equivalence checking approach we were able to detect as well as locate the source of the bug quickly. Moreover, after fixing the bug in the described way both models were equivalent. Please note that the TLM model did not suffer from this bug because it uses `memcpy` to read a 32 bit value from the memory (analogously to line 10 in Figure 2).

3.3.2 TLM and RTL model

The RTL model of the Y86 CPU also behaved differently from the TLM model. Simple sequential programs worked fine. However, when more complex programs using the jump instruction have been executed the equivalence check failed. More precisely, the RTL model never stopped running and seemed being stuck in a loop. The equivalence

report revealed a lot of unmatched accesses to the *Instruction Pointer* (IP)[4]. Using the verbose output mode of the equivalence checking server we identified the first unmatched IP access in the RTL and TLM model just during the processing of a jump-instruction[5]. This was possible because in verbose output mode the file and line of the location of the unmatched variable access is displayed. The output was:

```
TLM: IP WRITE 1b y86_tlm/y86.cpp:237
RTL: IP WRITE 19 y86_rtl/y86_cpu.cc:246
```

The TLM model wrote the value `1b` to the IP variable, while the RTL model wrote `19`. The file and location indicate where the variable access was reported. After examining the function to process a jump in both models (the returned line number is within the respective code of the function) the error was located in the computation of the target address in the RTL model: When the RTL model computed the target address after a jump, it did not add the length of the jump instruction (2 bytes) to the address, whereas the TLM model (and the ISA model) added the length. This target address is stored in the IP. The above example clearly shows that the RTL model did not increase the IP by 2 (the length of the jump instruction). After fixing this bug the equivalence report showed no unmatched variable accesses and the RTL model terminated correctly for all test programs.

In summary, using the proposed equivalence checking framework also the bug and its location was found very fast. In particular the verbose mode supported the localization.

In the following the efficiency of the approach is considered.

3.4 Performance Analysis

The equivalence checking server uses multiple threads and can handle millions of variable accesses. In Figure 6 the run-time of a complete equivalence check between the Y86 TLM and RTL model is shown. We performed the experiments on a Core 2 Duo 2.8 GHz computer with 4 GB of main memory. The more variables are chosen to be compared in the equivalence check, the longer is the run-time. Since the check compares two models and hence we implemented separate threads to handle the incoming data, the speed-up using a dual-core CPU is within a factor of two in comparison to a single core CPU. When using a CPU with more than two cores a further speed-up can be expected because a separate thread per variable is invoked.

4. CONCLUSIONS

In this paper we have presented an equivalence checking framework for ESL design flows. The framework has been implemented as a library following a client-server architecture. The clients are the SystemC models checked for equivalence. They communicate with the server via sockets. For equivalence checking read and write accesses of user selected variables are reported to the server using the developed library. If all reported accesses can be matched, the SystemC models have the same behavior.

In summary, our approach improves the design productivity since less effort is needed for setting up and performing equivalence checking. We have also shown that the multi-threaded architecture results in scalable equivalence verification. Moreover, in case of non-equivalent behavior the time required for debugging is reduced because bugs can be detected and located quickly with the verbose mode.

Several directions are left for future work. We plan to develop methods supporting the process of selecting variables which have to be compared during the equivalence check. Moreover, we want to investigate how our methods can be combined with coverage techniques including functional as well as code coverage techniques. As an example, the test-cases need to be improved if based on the performed model extension a certain (read or write) variable access is never reported. Furthermore, we plan to integrate an interactive debugging mode which uses a rollback mechanism for the simulation such that more data in the context of an unmatched access can be analyzed.

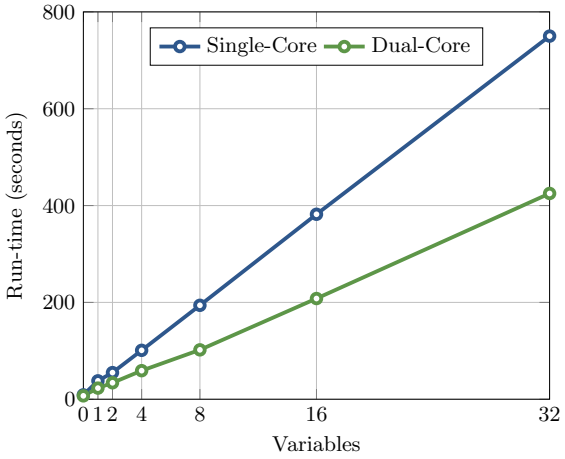

Figure 6: Equivalence check run-time

[4]This register points to the address of the next instruction. Please note that the length of an instruction is between 1 to 3 bytes.

[5]The jump-instruction `jnz dist` adds the given offset – termed distance – to the IP if the zero flag has not been set.

References

[1] Standard template library programmer's guide. http://www.sgi.com/tech/stl/.

[2] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.

[3] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. SystemC cosimulation and emulation of multiprocessor SoC designs. *Computer*, 36(4):53–59, 2003.

[4] A. Biere, D. Kroening, G. Weissenbacher, and C. Wintersteiger. *Digitaltechnik - eine praxisnahe Einführung*. Springer, 2008.

[5] N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva. Towards equivalence checking between TLM and RTL models. In *MEMOCODE*, pages 113–122, 2007.

[6] L. Cai and D. Gajski. Transaction level modeling: an overview. In *CODES+ISSS*, pages 19–24, 2003.

[7] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma. Non-cycle-accurate sequential equivalence checking. In *DAC*, pages 460–465, 2009.

[8] M. Fujita. Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths. *ACM Trans. Design Autom. Electr. Syst.*, 10(4):610–626, 2005.

[9] F. Fummi, M. Loghi, G. Perbellini, and M. Poncino. SystemC co-simulation for core-based embedded systems. *Design Automation for Embedded Systems*, 11(2):141–166, 2007.

[10] P. Georgelin and V. Krishnaswamy. Towards a C++-based design methodology facilitating sequential equivalence checking. In *DAC*, pages 93–96, 2006.

[11] F. Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2006.

[12] E. I. Goldberg, M. R. Prasad, and R. K. Brayton. Using SAT for combinational equivalence checking. In *DATE*, pages 114–121, 2001.

[13] D. Große and R. Drechsler. *Quality-Driven SystemC Design*. Springer, 2010.

[14] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[15] IEEE Std. 1666. *IEEE Standard SystemC Language Reference Manual*, 2005.

[16] Intel Corporation. *IA-32 Architecture Software Developer's Manual*, 2003.

[17] A. Koelbl, J. R. Burch, and C. Pixley. Memory modeling in ESL-RTL equivalence checking. In *DAC*, pages 205–209, 2007.

[18] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371, 2003.

[19] M. N. Mneimneh and K. A. Sakallah. Principles of sequential-equivalence verification. *IEEE Design & Test of Comp.*, 22(3):248–257, 2005.

[20] Open SystemC Initiative (OSCI). *www.systemc.org*.

[21] Open SystemC Initiative (OSCI). *OSCI TLM-2.0 User Manual*, 2008.

[22] J. Park, B. Lee, K. Lim, J. Kim, S. Kim, and K.-H. Baek. Co-simulation of SystemC TLM with RTL HDL for surveillance camera system verification. In *IEEE International Conference on Electronics, Circuits and Systems*, pages 474 –477, 2008.

[23] S. Vasudevan, J. A. Abraham, V. Viswanath, and J. Tu. Automatic decomposition for sequential equivalence checking of system level and RTL descriptions. In *MEMOCODE*, pages 71–80, 2006.