A Framework for Exploration of Parallel SystemC Simulation on the Single-chip Cloud Computer

Christoph Roth, Simon Reder, Oliver Sander, Michael Hübner, Jürgen Becker
Karlsruhe Institute of Technology (KIT)
Institute for Information Processing Technology (ITIV)
Engesserstraße 5
76131 Karlsruhe, Germany
{christoph.roth, oliver.sander, michael.huebner, becker}@kit.edu

ABSTRACT

Since the number of cores integrated on a single die is expected to increase steadily, new memory and communication architectures as well as programming methods are needed and currently explored by manufacturers. In this context, Intel Labs developed the Single-chip Cloud Computer (SCC), a 48-core experimental processor, serving as a platform for many-core software research. Within this paper a framework targeting the investigation of SystemC kernel parallelization on the SCC is presented. The framework provides the basis for implementation of different synchronization schemes while combining distributed and shared memory programming models and exploiting multiple distinct address spaces. As a case study, a synchronous parallelization scheme is preliminarily evaluated by means of several simulation models of different accuracy. Results of the analysis give a first evidence of the applicability of the synchronous parallelization method on the homogeneous non-cache coherent manycore architecture of the SCC for detailed system simulation.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Hardware description languages, Simulation*; I.6.8 [**Simulation and Modelling**]: Types of Simulation—*Discrete event, Distributed, Parallel*

General Terms

Design Languages

Keywords

Single-chip Cloud Computer, Parallel Simulation, SystemC

1. INTRODUCTION

Today, modern CPUs already consist of ten and more cores. It is expected, that the number of cores per die steadily grows in the next few years. This trend from multicore towards manycore chips is driven by the need to optimize performance per watt which results in an increase of parallelism instead of the clock frequency. Currently, it is not known, how future manycore architectures will exactly look like. They can be homogeneous or heterogeneous, generic or application-specific [15].

With increasing number of cores, the demand for communication between the cores also increases. As a result, communication time rapidly becomes the dominant factor and outweighs the time it takes for computation on the cores which strongly reduces performance. It is even possible, that execution performance of an application distributed over several cores becomes worse than performance of sequential execution on a single core. Because of that, improvements of the memory and communication architecture on hardware as well as on software level can significantly enhance the overall performance of an application.

The Single-chip Cloud Computer was developed in order to provide a software research platform for future manycore architectures [5]. The SCC is a homgeneous manycore chip that supports distributed as well as shared memory programming models. Providing a network-on-chip based on fast non-cache coherent shared memory, it changes the traditional conditions of cache-coherent shared memory being the only reasonable approach for exploiting fine-grain parallelism instead of message passing.

In this work, we propose a SystemC [2] runtime environment for the Single-chip Cloud Computer that allows investigating programming techniques for parallel system simulation on the SCC manycore platform. Different parallel simulation architectures and algorithms can be implemented on top of a basic simulator template that abstracts from the underlying hardware and software communication libraries. By means of a case study we demonstrate applicability of the developed framework. We therefore followed a pragmatic approach and transferred a synchronous system simulation architecture that works well on traditional cache-coherent machines to the non-cache coherent domain of the SCC. Such a synchronous architecture in combination with detailed system models usually connotes high synchronization overhead. From this point of view, detailed system simulation can be regarded as a worst case scenario with respect to performance evaluation of the SCC. Analysis results show, that a lot of SystemC and SCC specific optimizations and manual adaptations are necessary in order to be able to exploit parallelism and accelerate system simulation on the

Simutools 2012, March 19-23, Desenzano del Garda, Italy

DOI 10.4108/icst.simutools.2012.247751

SCC using the chosen synchronous approach.

The remainder of this paper is organized as follows: In section 2 we summarize a selection of related work. Fundamentals, including a description of the sequential OSCI SystemC kernel as well as the SCC platform are presented in section 3. The framework architecture is detailed in section 4. A case study exploiting the framework for a full synchronous simulation is presented in section 5. Finally, in section 6 we conclude and give an outlook to further work.

2. RELATED WORK

Within the last years, diverse approaches for parallelization of the SystemC kernel have been proposed. Most of them target distributed execution on a workstation cluster [8][9][14][10] or parallel execution on symmetric multiprocessor (SMP) hosts [18][20]. Other approaches try to exploit specialized hardware architectures like the Cell processor [13]. One major problem that all these examples are faced with, is the synchronization between different processing elements that execute parts of the model in parallel, also known as the causality problem [12]. To cope with the causality problem various algorithms have been proposed. They can be divided into *conservative* and *optimistic* approaches. In short, conservative synchronization algorithms avoid violating the causality relationships between logical processes (LP). They always guarantee events to be delivered in the correct time order. In contrast, optimistic approaches allow violating the causality relationships but provide rollback mechanisms to restore already past points in time. Conservative stategies in turn can be *asynchronous* or *synchronous*. In asynchronous approaches each LP maintains a local time. Depending on the time stamps received from all neighboring LPs each LP determines the next safe point in time until which it is allowed to execute. Instead, in synchronous approaches logical processes regularly synchronize globally via barriers to determine the next point in simulation time until which events can be processed safely.

The runtime environment proposed in this paper focusses on the investigation of conservative synchronous and asynchronous approaches in order to evaluate their applicability in the context of detailed system simulation on the Single-chip Cloud Computer from Intel. To the best of our knowledge, this work is the first one reporting about a framework for investigation of parallel system simulation on the SCC.

3. FUNDAMENTALS

3.1 OSCI SystemC Kernel

A SystemC simulation is a so called discrete-event simulation. The behaviour of a system over time is modeled by logical processes generating discrete events which in turn can trigger the execution of the same or other logical processes and change the actual system state. The execution of the simulation is controlled by the simulation kernel that handles synchronization as well as process scheduling.

In general, during execution the kernel runs through different phases called *Elaboration Phase*, *Initialization Phase*, *Evaluation Phase*, *Update Phase*, *Delta Notification Phase* and *Timed Notification Phase*. Except *Elaboration Phase* and *Initialization Phase* these are executed in a loop. The functional principle is illustrated in Listing 1.

First, during *Elaboration Phase* and *Initialization Phase* data structures are initialized, connectivity between modules is established and all processes are made runnable and are randomly invoked. In the *Evaluation Phase*, all runnable processes are executed sequentially. The processes possibly generate *immediate notifications* which cause processes to become runnable again. If no more runnable processes exist, the kernel switches to the *Update Phase* in which all update requests on channels are processed. These were generated during the *Evaluation Phase* when a process wrote a new value in a channel. The updating of channels possibly causes the generation of *delayed notifications* during the *Delta Notification Phase* and effects a return to the *Evaluation Phase* for process execution. *Delayed notifications* prevent processes from recognising new values immediately, but only after the next so called *delta cycle* [2]. If no *delayed notifications* exist the kernel switches to the *Timed Notification Phase*. Simulation time is advanced to the next pending *timed notification*. Processes sensitive to the current time are made runnable. Afterwards, the kernel returns to the *Evaluation Phase*. Simulation ends if no more *timed notifications* exist that could trigger a process.

Algorithm 1 SystemC Simulation Loop

```
1: elaboration and initialization
2: while timed notifications exist do
3:     while runnable processes exist do
4:         while runnable processes exist do
5:             select and execute a runnable process
6:             process immediate notifications
7:         end while
8:         process update requests
9:         if delta notifications exist then
10:            process delta notifications
11:        end if
12:    end while
13:    advance simulation time
14:    process timed notifications
15: end while
```

3.2 Single-chip Cloud Computer Overview

The SCC is a many-core CPU consisting of 24 tiles connected by a 6 by 4 mesh network (Fig. 1). Each tile contains two second generation P54C cores [11], 16 KB instruction and data L1 caches per core, a unified 256 KB L2 cache, a Mesh Interface Unit (MIU), a 16 KB Message Passing Buffer (MPB) and two test-and-set registers. Via the MIU each tile is connected to a router which integrates the tiles into the mesh. The MIU packetizes data onto the mesh and depacketizes data from the mesh using a round-robin scheme to arbitrate between the two cores on the tile. [3][16].

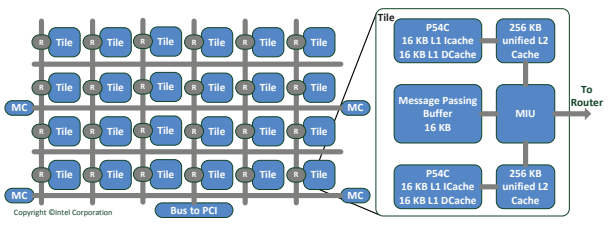

Figure 1: SCC Architecture (source: [3])

The memory architecture of the SCC is composed of multiple distinct address spaces and supports both, distributed and shared memory programming models. In general, different configurations are possible. Memory regions are configured by the help of lookup tables (LUT) which translate the physical 32 bit address of the cores to a global system address. Thereby, the LUT configuration determines whether a physical address refers to off-chip DDR3 memory or to the on-die MPB memory [3]. Within this work the following configuration is used:

- *Private off-chip DRAM:* External memory that corresponds to the main memory of a conventional PC. Each core has its own region. Memory access is performed via four DDR3 memory controllers, L2 and L1 caches.
- *Shared off-chip DRAM:* This type of external memory can be shared among cores. It is also called external shared memory (ESHM). It is generally configurable as non-cachable or cachable. In case of cachable ESHM, cache coherency is only managaged by software which appeared to be very slow. Because of that, we only use non-cachable ESHM.
- *Shared on-chip SRAM:* This type is intended for fast on-chip data exchange between cores. It is based on the MPB. The SCC does not offer any hardware managed L2 cache coherency. Communication is performed by transferring data from private memory and L1 cache of the sending core via MPB to the L1 cache of the destination core. L1 caching is possible due to the featuring of a new memory type called Message Passing Buffer Type (MPBT).

4. FRAMEWORK CONCEPT

For parallel system simulation on the SCC with our framework, each SCC core taking part in a simulation must execute an instance of a runtime environment. Each instance executes specific parts of the SystemC kernel, depending on the method of parallelization. E.g. it is imaginable that each instance executes the complete SystemC kernel including all the phases mentioned in section 3.1. This would correspond to the approach presented in [8]. Another attempt would be to provide different types of runtime environments and to execute only specific phases within each instance. This approach is applied in [18] and is also implemented in the case study in section 5.

Generally, the runtime environment is executed on top of the SCC operating system, a modified Linux Kernel 2.6.16 [16]. The overall software structure of the runtime environment is illustrated in Fig. 2. In order to make the framework code reusable we chose a hierarchical architecture consisting of three abstraction levels, each fulfilling specific tasks. By using inheritance functionality of the lower levels is provided to the upper levels. Beside that, different manifestations of a parallel SystemC simulator requiring different communication and synchronization mechanisms can be build.

On the first abstraction level called *Base Application Level* basic SCC specific routines for memory management, message passing via MPB and synchronization of shared memory access are made available to the upper layers. The provided message passing routines incorporate the so called RCCE library [17] which is a small communication library

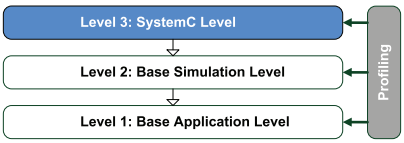

Figure 2: Framework Architecture

specialized for the SCC architecture. The second abstraction level is called *Base Simulation Level* and provides all fundamental functionality that is necessary to realize parallel discrete event simulation [12] on the SCC. Therefore this layer allows implementing different synchronization strategies using synchronous or asynchronous time management. E.g. in case of a pure synchronous strategy the *Base Simulation Level* must provide a time management based on global barrier synchronization. In case of a pure asynchronous strategy the *Base Simulation Level* e.g. implements the *Chandy/Misa/Bryant Null Message Algorithm* [7]. Finally, on the topmost layer (*SystemC Layer*) all SystemC kernel and model specific functionality is integrated into the runtime environment. The implementation is based on the OSCI SystemC reference kernel [2]. The entry point into the kernel is given by its *sc_simcontext* class. In order to allow simulation control from outside, relevant classes and methods need to be adapted to the chosen parallelization strategy.

The runtime environment can be executed in a profiling mode which allows tracking timing characteristics and performance bottlenecks on all three abstraction levels. For that reason a profiling API exists that allows instrumenting the code by the programer. In the now following case study we show how this simulation framework can be adapted to a specific use case.

5. CASE STUDY

5.1 Synchronous Parallel SystemC Kernel

Full synchronous execution is mentioned in the literature [4] as the most obvious event driven approach. The overall parallel simulation advances in a lock-step fashion. The simulated time is common accross all processors. According to [4] barriers are the easiest way to determine time step completion. Because of that, we followed a pragmatic approach and transferred a full synchronous method that works well on cache coherent machines [20][18] to the non-cache coherent domain. The scheme is based on the fact that according to the OSCI SystemC standard the order in which runnable processes are executed is not predefined during a delta cycle. This can be exploited for parallel execution due to the causal indenpendency of SystemC processes during a delta cycle. The overall approach is illustrated in Fig. 3. Processors are divided into master and workers. The master processor distributes computation tasks among worker processors. Serial and parallel phases are separated by barrier primitives. In principle, this parallelization concept requires all persistent data associated with the simulation model to be shared among all participating processors. On cache coherent SMP machines this is usually done via cached shared memory, being the fastest variant. On the non-cache coherent SCC one can chose between the already mentioned off-chip shared memory (ESHM) and on-chip shared memory within the MPB.

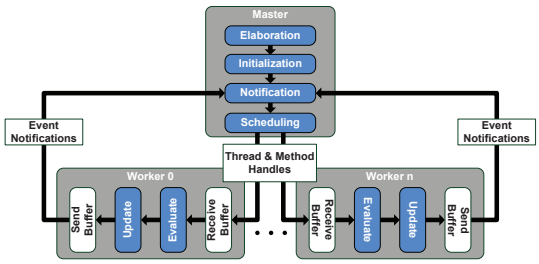

Figure 3: Synchronous Parallel SystemC Kernel

From the framework point of view, level 2 and 3 need to be adapted to behave as either master or worker. Both implement an barrier based event system on level 2. Thereby, time advancement is controlled by the master. The event system of the workers is extended with a transmission routine, that forwards notifications and dynamic sensitivity changes to the master processor. This extension of the event system is implemented using the on-chip shared memory within the MPB instead of the slower ESHM on level 1. Based on the received event notifications, the master is able to determine the active processes for the next delta cycle. We use an additional receive buffer within the MPB for each worker processor, which contains identifiers for the processes the workers have to evaluate. The master fills the queues of the workers with processes and does basic centralized load balancing using a uniform load balancing policy. All remaining SystemC module and channel data that needs to be shared is stored within the ESHM. Alternatives would be either a broadcast or a point to point transmission of signal changes between processes assigned to different cores. The former would overload the MPB, the latter would imply high overhead in every slave for determining the communication partner.

5.2 Optimizations

5.2.1 FIFO Queues

Of particular importance for efficiency is the implementation of the MPB communication mechanism between the cores on level 1. By default the RCCE library only provides a blocking communication mechanism which drastically reduces performance of our use case. Because of that, we implemented FIFO message queues which allow non-blocking communication. Beside that, we extended the message passing functionality of the RCCE library by a buffer in the private memory of the sender side. The buffers in the private memory allow storing a configurable amount of data before transmission is triggered via the MPB which reduces the frequency of MPB access.

5.2.2 Double Buffering

To limit the number of read and write operations on channel data that needs to be shared, we generally pursue a double buffer strategy for all SystemC channels that support the evaluate/update paradigm [2]. During evaluation phase, we read the value of the channels only once in a delta cycle from ESHM and copy it into a local buffer. SystemC processes are generally executed on this local copy of the channel data which means, every read and write operation on the channels during a delta cycle is done using the local memory. Finally, after evaluation, the global update phase follows during which local updates are written back into ESHM. This procedure is possible, since the current output value of evaluate/update channels does not change during evaluation. As a whole, we take advantage of an increased data locality and decrease synchronization overhead during channel access. To make sure that module members, that need to be shared also work properly, we implemented a template class on level 3 to wrap around the original member types. Overloading the C++ operators for conversion and assignment allows almost the same syntax as the original type does, but offers the possibility to use the corresponding shared buffer on level 1 instead of the local copy.

5.2.3 Static Binding

The data of the SystemC model itself mainly consists of the instances of *sc_module*, *sc_threads* and the channel instances which are both created during elaboration. *sc_thread* processes, which are basically coroutines (or user threads) with a separate program stack, require the stack to be stored permanently. This stack data does not need to be shared if the *sc_thread* is always evaluated on the same core. Since our measurements revealed large communication overhead when treating *sc_threads* dynamically, we bind each *sc_threads* statically to one of the cores, what consequently means disabling any kind of load balancing for *sc_threads*. The master distributes *sc_thread* processes first, so that there are mainly *sc_method* processes left when the first workers run out of processes. Regarding *sc_method* we leave the option of dynamic scheduling and static binding. Furthermore, in the dynamic case work-stealing can be successfully applied which enables workers to take process identifiers from other queues in case they run out of processes. Static binding is an optimization on level 3.

5.2.4 Manual Grouping

It is possible to further increase simulation performance based on knowledge of the model to be simulated. Therefore, we implemented a way to manually group processes together, so that the group of processes is bound statically (similar to the static binding of *sc_threads*) to the same processor. Thus, shared data and channel data that are common to processes of the same group can be stored locally in the local space of the corresponding processor. Of course, this level 3 optimization is mainly reasonable if processes are in a tight communication relationship to each other. However, the described way of grouping is not limited to processes of a single module. E.g. by grouping the processes of several modules together, it is possible to use channel implementations based on private memory for all channels connecting those modules.

5.2.5 Special Treatment of the Clock Signal

As described above, synchronization is generally performed on the delta cycle level, which means, that a complete iteration through evaluation, update and notification phases is done during which global synchronization must be enforced three times. When thinking of clock signals this fact can decrease simulation performance, since for each generated timed clock event an additional delta cycle is generated exclusively for execution of the thread that changes the clock. Only afterwards, clock synchronous processes are notified for execution. Because of that, we applied a modification on level 3 and changed the *sc_clock* implementation such that a

clock event does not generate a delta cycle but an immediate notification on all clock synchronous processes. That way, we save one delta cycle (one global synchronization cycle) per simulated clock cycle. During our evaluations this resulted in an increase of performance of about 8% for smaller models.

5.3 Performance Evaluation

5.3.1 Synthetic Scenario

By means of a synthetic model we examined basic performance boundaries of the synchronous parallel kernel. Due to the multi-dimensional exploration space of the SCC we constrained the evaluation to the simple case of one master and only two workers being located to each other as close as possible. The structure of the used synthetic model is illustrated in Fig. 5. It consists of two modules *component_0* and *component_1* interconnected by m signals of type $sc\_uint < 8 >$. Each component in turn contains n modules called *stages*. The model is synchronous. With every clock cycle each stage performs a configurable number of floating point operations (*complexity*), increments input signal values and afterwards transfers these values to its respective output signals. That way, the whole model represents a configurable pipeline.

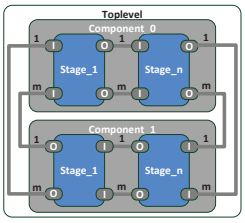

Figure 5: Synthetic Model

Fig.4 illustrates the speedup of parallel compared to sequential execution on a single core when varying different parameters. Considering case A and B it is apparent that chosing either *sc_thread* or *sc_method* as well as applying either dynamic task mapping or not, decides on the course of the speedup curve. The synchronous parallel kernel particularly scales better in case of a high amount of signals combined with a fix task mapping and manual grouping (see case B) due to the possibility to store component-internal signals locally. Comparing *sc_thread* and *sc_method* in case of fixed mapping and manual grouping *sc_thread* reveals an additional gain, although threads hold a separate stack which must be fetched/stored from/in memory in case of a context switch. The reason is that context switching of threads is performed by the workers in parallel too during the evaluation phase. Case C illustrates scalability depending on the number of stages/threads per core. we used fix mapping in combination with manual grouping. After reaching a maximum possible speedup, performance again decreases. The master becomes the bottleneck since with increasing thread number the master must process an increasing number of event notifications.

5.3.2 MPSoC Model

In the second experiment we examined execution performance when simulating different MPSoC models of varying accuracy. First, we investigated performance when using the abstract but deterministic NoC model presented in [19] together with cycle accurate Plasma processors [1] as processing elements. In a second step we made recourse to the HeMPS MPSoC [6] which is a full accurate MPSoC model consisting of Plasma cores interconnected by the so called Hermes-NoC. In either case, the software running on the Plasmas repetitively performs some dummy calculations before the next destination node is addressed one after another and a packet is transmitted. For our measurements we applied the optimizations mentioned in section 5.2. Results of performance analysis when using the mixed abstraction level model are illustrated in Fig. 6. The speedup compared to the sequential case is shown.

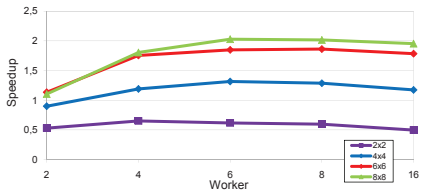

Figure 6: Mixed Abstraction Levels

As can be seen, the maximum reachable speedup first increases with increasing NoC size and number of workers until reaching a maximum (e.g. 2.02 for 8x8 NoC and 6 workers). However, for a 2x2 NoC the computational overhead compared to the synchronization overhead is too small to get a speedup > 1. The additional gain decreases with increasing model size which is caused by the master node that becomes a bottleneck due to an increasing number of event notifications. Furthermore, after reaching the maximum the curves descend with increasing number of workers due to increasing communication between cores executing parts of the model. Fig. 7 illustrates performance results when simulating a 2x2 as well as a 4x4 HeMPS system. Similar to the mixed abstraction level MPSoC the maximum achievable speedup is strongly limited whereby the 4x4 RTL network expectably does slightly worse.

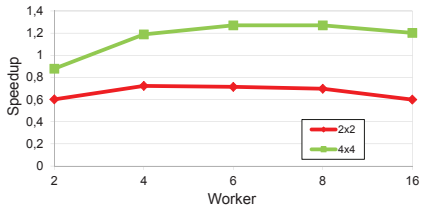

Figure 7: Register Transfer Level

5.4 Short Discussion

As has been demonstrated, a lot of manual adaptations on all levels (1 to 3) are necessary in order to accelerate detailed system simulation using the described synchronous implementation. Reasons for the limited maximum speedup are on the one hand, the frequent but slow access to the model data located in the external shared memory and on the other hand the immense number of event notifications and sensitivities that need to be exchanged between master and workers (even if events are purely local) making communication between master and workers to a great bottleneck. Because of that, our future investigations regarding the presented case study comprises a deep analysis and profiling of further possible variants of the current implementation, e.g.

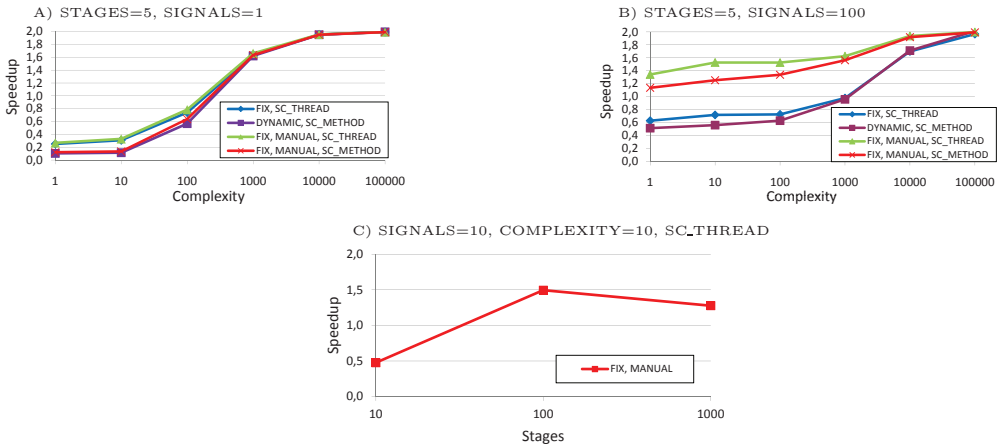

Figure 4: Results of the Synthetic Scenario

using different load balancing policies and mappings strategies. Our final goal is the comparison of other approaches like the one presented in [8] in order to find the best solution for the SCC. Also investigations of asynchronous or hybrid approaches are imaginable. Using the approach of [8] we expect at least a reduction of the number of event notifications that need to be transmitted accross the network since only notifications generated by channels interconnecting different partitions need to be exchanged here.

6. CONCLUSION

Within this work we presented a framework that serves as starting point and basis for research in the context of detailed system simulation on the Intel SCC. Due to its extensible design the framework allows implementing different communication and synchronization strategies. Within this work we demonstrated applicability of the framework by means of a case study that focussed on full synchronous parallel execution. By the help of this case study we also got a first insight into performance limits of the implemented synchronous strategy. Our future work comprises the optimization of the presented full synchronous method. Furthermore, we want to compare the presented approach to other synchronous and asynchronous approaches or even combine them in oder to find the best solution.

7. REFERENCES

[1] http://www.opencores.org.

[2] IEEE Standard System C Language Reference Manual. *IEEE Std 1666-2005)*, pages 1 –423, aug. 2006.

[3] SCC External Architecture Specification (EAS) Revision 1.1. 2010.

[4] M. L. Bailey, J. V. Briner, Jr., and R. D. Chamberlain. Parallel logic simulation of VLSI systems. *ACM Comput. Surv.*, 26:255–294, September 1994.

[5] M. Baron. The Single-Chip Cloud Computer. 2010.

[6] E. Carara, R. de Oliveira, N. Calazans, and F. Moraes. HeMPS - a framework for NoC-based MPSoC generation. In *ISCAS 2009*.

[7] M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of distributed Programs. In *IEEE Transactions on Software Engineering SE-5, (5)*, pages 440–452, 1979.

[8] B. Chopard, P. Combes, and J. Zory. A Conservative Approach to SystemC Parallelization. In *Computational Science ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 653–660. Springer Berlin / Heidelberg, 2006.

[9] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory. Relaxing Synchronization in a Parallel SystemC Kernel. pages 180 –187, 2008.

[10] D. Cox. RITSim: Distributed SystemC simulation. In *Master thesis, Rochester Institute of Technology*, 2005.

[11] T. D. Anderson. Pentium Processor System Architecture. *Addison Wesley*, 1995.

[12] R. M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[13] D. Houzet, S. Huet, and L. Kaouane. SysCellC: SystemC on Cell. In *Computational Science and Applications - ICCSA'2008*, page 574. IEEE, 2008.

[14] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele. Scalably distributed SystemC simulation for embedded applications. pages 271 –274, jun. 2008.

[15] R. Leupers, L. Eeckhout, G. Martin, F. Schirrmeister, N. Topham, and X. Chen. Virtual Manycore platforms: Moving towards 100+ processor cores. In *DATE 2011*, pages 1 –6, march 2011.

[16] T. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: the Programmer's View. In *IEEE/ACM SC'2010*.

[17] T. Mattson and R. van der Wijngaart. RCCE: a Small Library for Many-Core Communication Software 1.0 - release. 2010.

[18] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi. Parallelizing systemc kernel for fast hardware simulation on smp machines. PADS '09, pages 80–87, Washington, DC, USA, 2009. IEEE Computer Society.

[19] C. Roth, O. Sander, M. Kühnle, and J. Becker. HLA-based Simulation Environment for Distributed SystemC Simulation. In *Simutools*, 2011.

[20] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann. parSC: synchronous parallel systemc simulation on multi-core host architectures. CODES/ISSS '10. ACM, 2010.