Parallel native-simulation for multi-processing embedded systems

Alejandro Nicolas
Microelectronics Engineering Group
University of Cantabria
Santander, Spain
nicolasa@teisa.unican.es

Pablo Sanchez
Microelectronics Engineering Group
University of Cantabria
Santander, Spain
sanchez@teisa.unican.es

***Abstract*—The number of cores in embedded systems is continuously growing, supporting increasingly complex concurrent applications. In order to verify that the systems comply specification requirements during the design process, fast simulations and performance analysis tools are required. These simulation frameworks typically use virtualization or host-compiled simulation techniques. On one hand, current host compiler simulators normally offer fast simulations, but they do not exploit host parallelism capacity. On the other hand, some virtual emulation frameworks take advantage of host parallelization, but they do not achieve simulations as fast as native (host-compiled) simulators because of the dynamic binary translation. This paper proposes a parallel host-compiled simulation methodology that aims to make an efficient use of multi-core host platforms. The performance of the proposed technique has been evaluated with the PARSEC benchmark suite [10]. The evaluation also includes comparisons with native execution and other parallel simulation tools. Results demonstrate that the proposed technique reduces simulation time and provide fast estimations of embedded SW code.**

Keywords—virtual platform; parallel host-compiled simulation; performance analysis; many-core embedded system

I. Introduction

Current embedded systems are designed using HW platforms with Multi-Processor Systems-on-Chip (MPSoC). Additionally, the technology evolution enables increasing the number of cores in these embedded platforms. Designing embedded applications for MPSoC is a very complex and costly task, and verifying the correctness of the complete system adds additional difficulties to the process. The embedded system designers normally have to address two challenges: selecting the target platform and obtaining maximum performance of embedded software applications. Moreover, these two tasks are interconnected which adds extra difficulty.

In order to provide estimations and metrics that guide design decisions, several performance analysis and simulation frameworks have been proposed. The estimation tools have to be fast and flexible, especially when wide design space explorations are required. They should also take into account the target embedded platform architecture, being able to offer execution time and power consumption estimations.

The target (simulated) platform could have a multi-core or many-core architecture. Target-platform aspects, among others, have to be taken into account during simulation.

The simulated target RTOS (tRTOS) schedules the execution of the target threads and processes in the target cores. These target threads/processes parallelize the embedded application. The main objective of this paper is to develop a performance analysis framework that can simulate very fast and efficiently this type of target system in a general-purpose computer (host platform). Host platforms normally integrate several multi-core processors with SMP (Symmetric Multi-Processing) capability and they will soon contain several tens of cores [12]. Thus, this paper proposes a performance analysis methodology and a virtual platform (Virtual Parallel platform for Performance Estimation, VIPPE) for simulating embedded parallel applications in target many-core platforms using a SMP multi-core host computer.

In native (host-compiled) simulation, the execution time of the embedded software in the target platform is estimated during host-code execution. The estimation functions take into account target core instructions, caches, buses and memory access times.

II. State of the Art

With the purpose of validating constraints and exploring different implementations in multi-/many-core platforms, designers require performance analysis frameworks capable of carrying out fast system simulations. For embedded system evaluation, several static performance analysis techniques have been proposed [1], although they have limited applicability. Alternatively, most performance analysis frameworks are based on simulation, which provides faster and more accurate results. During the last years, two types of simulation techniques have been proposed.

The first type executes target-compiled applications on host models of the target platforms. For example, they could use an Instruction Set Simulator (ISS) to model the target cores [2]. These tools provide very accurate results but poor simulation

978-1-4673-8035-5/15 $31.00 © 2015 IEEE
DOI 10.1109/DSD.2015.75

CPS Conference Publishing Services

performances. Other approaches use binary-translation techniques to improve simulation performances [3]. However, these solutions are limited to sequential simulation and they are not able to take advantage of parallelism. Currently, this is a great limitation since the applications tend to be concurrent in order to exploit parallelism on target platforms. To face this issue, parallel simulation solutions based on QEMU have been proposed, such as COREMU [11]. These solutions offer good performances regarding parallel speed-up but have limitations in simulation time due to binary translation. In addition, the parallelism and speed-up is limited to the number of cores of the target (simulated) platform.

The second type of performance analysis framework is based on host-compiled (native) simulation. This solution offer faster simulations than previous techniques since it avoids binary translation. However these solutions typically provide less accurate results than binary translation based techniques. In host-compiled simulation, the embedded software is compiled in the host computer with some additional instrumentation code and/or support libraries (e.g. the Transaction-Level Modeling kernel of SystemC [4]). It could also include models of the target RTOS [5]. Nevertheless, there is a growing interest on parallel implementations of host-compiled simulators during the last years. The synchronization strategy of these simulators could be synchronous [6] (if there are global synchronization points or clocks) or asynchronous [7]. The last strategy improves simulation performance although the implementation is more complex. A conservative simulator guarantees that there are not causality errors while the optimistic approach in [8] could have some errors that force the simulator to recover a previous state.

To reduce simulation time, this paper presents a conservative parallel host-compiled simulation technique with asynchronous synchronization strategy. The goal is to face poor simulations performance of DBT simulators and the parallel evolutions appeared such us [11]. For such purpose, this works relies on host-compiled technique. In addition, the simulator solves the problem of not parallelization supported in host-compiled simulator such us [8]. Moreover, the parallelization performance of this technique of simulation is not limited by the number of virtual platform cores but the host platform cores.

III. HOST SIMULATION TECHNOLOGY

The host simulation technology consists of four stages.

A. *API provided by the simulator*

The simulator provides an optimal reduced set of primitives which make use of the services offered by the kernel.

B. *Instrumentation*

The instrumentation is automatically performed by using the LLVM framework [9]. This instrumentation brings the information necessary to provide several estimations (e.g. simulated execution time and number of memory accesses).

Once binaries from functional code are generated, they are linked together with the simulator libraries in order to obtain a final binary executable. This executable will run in the host platform to provide the estimation metrics.

C. *Target Platform Description*

The target platform description is taken as input for the binary executable obtained from compiling the user code sources and linking with the simulator libraries. This input description is read from XML files.

D. *Simulation Technique*

The main goal of the simulation technique is to exploit the concurrency of the application and to take advantage of the parallel capacity in the host environment. There are some terms that need to be introduced before the explanation of simulation technique: kernel thread is the thread master of the simulator; target threads are the required threads to be created explicitly by the user application code. Host threads are all the threads created by the simulator.

The simulation starts by the simulator environment initializations such as, for example, the target platform reading and its virtual initialization. Kernel thread performs these initializations. Then, kernel thread will launch the simulation mapping the application in a new host thread. The simulation is performed by mapping each target threads (t-threads on Fig. 1) to a host simulation thread (h-threads on Fig. 1).

Target threads shares their information related to code annotation with simulator kernel thread and this one is in charge of simulating the system through this information. The communication between target threads and kernel thread is implemented by shared variables. Kernel thread writes and reads atomically on shared variables and it does not lock target threads. Thus, target threads only need to be locked when they access to synchronization elements for reading, because accesses for writing are stored in a queue of events. The events of the queue are resolved by kernel thread when updates threads timings and unlock target threads. This fact minimizes locking and improves parallel execution.

So the proposed simulation technique intends to exploit host parallel capacity, simulating several application threads in parallel.

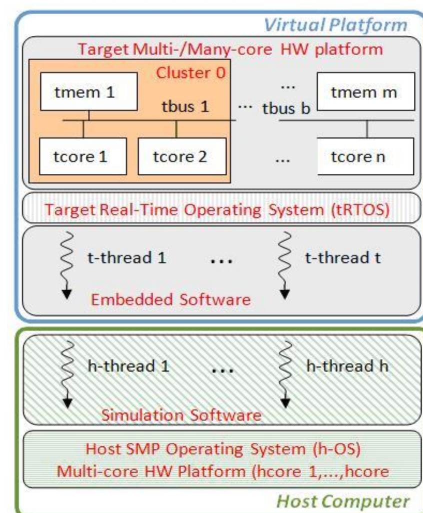

Fig. 1. Simulation of virtual platform

TABLE I. KERNEL ALGORITHM

1. - Begin simulation cycle. AddTime=0;
2. - While AddTime is less than a time slice:
   2.1. - Atomic read of thread local values
   2.2. - Select the threads with SimStatus="active" that will be scheduled to target cores
   2.3. - Check if all selected threads can be allocated. If not.
      2.3.1. - Return to the host control
      2.3.2. - Update thread local values and go to 2.3.
   2.4. - New_AddTime=minimum of the times to allocate of the selected threads. Estimate bus bandwidth.
   2.5. - Schedule threads until New_AddTime. Update shared variables and synchronization elements.
   2.6. - Check selected threads with status="locked".
      2.6.1- If they change their status to "active", go to 2.3.1.
      2.6.2- If they still locked, select a new thread to be schedule in the target core.
   2.7. - AddTime=AddTime+New_AddTime. Go to 2.3.
3. - Check all threads that do not have SimStatus=active. The new current values of shared variables and synchronization elements could modify their status.
4. - It simulation has not finished go to 1.

1) *Simulation kernel algorithm*

The simulation kernel thread handles the physical simulation time or global time (globalSimTime). Additionally, it maintains a list with the current values of all shared variables and synchronization elements (currentValueList). The list also includes information about the last global time in which every shared variable and synchronization element was updated.

TABLE I describes the kernel simulator algorithm. Kernel thread computes in parallel with the user code mapped on host threads. Kernel thread is in charge of scheduling the simulation on the virtual target platform. The simulation kernel reads atomically the target threads local time values and updates the kernel times values (atomic read operation) for each target thread in the simulation.

IV. EXPERIMENTS AND RESULTS

This section has two goals: evaluating the simulator performance comparing it with available tools in the state of the art such as COREMU and analyzing maximum possible speed up. For such purpose, applications from a parallel application benchmark, the PARSEC benchmark, were simulated in order to obtain simulation performance and study the limitations of the proposed approach.

All evaluation results were performed on a host with 8 Intel Xeon E5-2687W at 3.10 GHz. Every processor has 8 cores, thus the host platform integrates 64 cores with SMP capability running Fedora-Linux with kernel 3.16.2-201. Also the computer has 64 GB of RAM and 20 MB of cache.

A. *Comparison with other approaches*

This section is focused on comparing our simulator with other available virtualization platform: COREMU, a parallel extension of QEMU.

For comparing that, some examples have been simulated in both simulators, considering a dual core as target platform. These comparatives were done in the same conditions (TABLE II and TABLE III).

On one hand, regarding execution times, VIPPE simulation obtains better performance due to the native execution in comparison with binary translation of COREMU. On the other hand, the obtained results show how the COREMU speed up converges when the number of threads is the same or greater than the number of target cores. By contrast, the speed up from VIPPE simulations is limited by the number of threads or the application characteristics (*x264* and *Blackscholes*, this will be discussed in the following section). Although the speed up does not offer as much gain as COREMU (due to the normalization to sequential case), the execution time continues being smaller.

B. *Comparison with ideal speed up*

This section tries to analyze VIPPE performance depending on the characteristics of the application to simulate.

In order to clarify the level of parallelization, the following graphic (see Fig. 2) show the achieved speed up. Besides, this speed up is compared to the ideal maximum speed up. This maximum speed up is calculated by executing the applications directly on the host (without the simulator), therefore, avoiding the overhead added by simulator when annotating the code and modeling the system.

The kernel simulation thread is in charge of unlocking the simulation target threads. Thus, when the target threads execution time between locks is small compared to the time the kernel thread takes to free these threads, the simulation execution time is penalized. This penalty will depend on the number of target threads and the number of synchronizations. Then, the penalty of locking affects the simulation time. Hence, the bottleneck of the simulator is in the kernel speed for freeing the locked threads. Applications with low ratios (total_synchronizations/total_instructions) achieve parallelization improvement, whereas, applications with higher ratios are penalized as the number of application threads increases. The maximum scalability is explained by the Universal Law of Computational Scalability [13].

TABLE II. COREMU EXECUTION TIMES (SECONDS)

Example \ Threads	1	2	4	8
Swaptions	110.81	58.91	59.12	62.54
Blackscholes	130.83	65.68	66.52	67.01
X264	268.12	148.33	149.48	151.23

TABLE III. VIPPE EXECUTION TIMES (SECONDS)

Example \ Threads	1	2	4	8
Swaptions	5.57	2.99	2.10	1.76
Blackscholes	3.30	2.05	1.81	1.73
X264	5.99	6.99	8.10	8.43

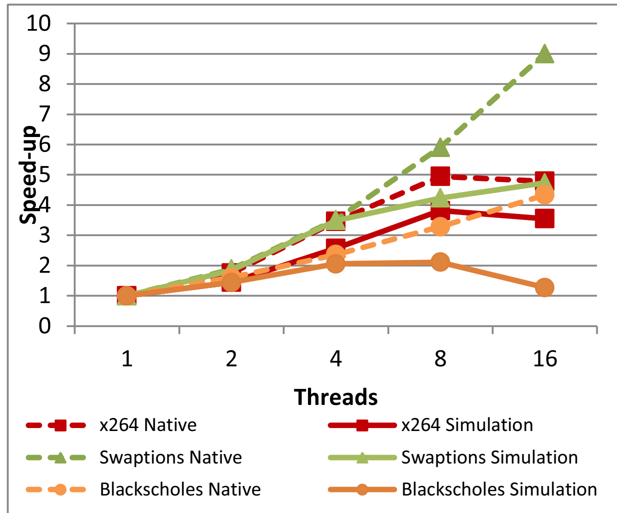

Fig. 2. Application simulation vs. native

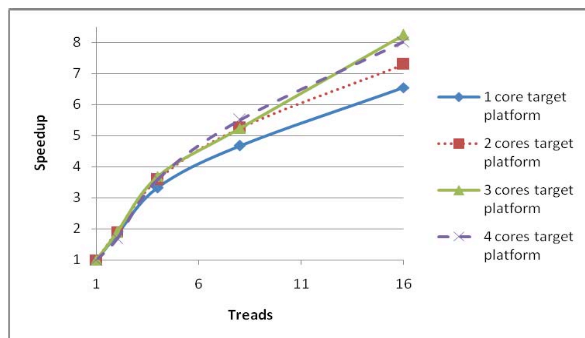

Fig. 3. Swaptions simulation time scalability when simulating different target platforms

VIPPE approach is independent from simulated platform and the scalability depends on the number of host cores. So it is possible to take full advantage of parallelization in host-cores almost independently of the target platform number of cores as can be seen in Fig. 3. This demonstrates that the target platform has not an important impact in the scalability of the simulation.

Finally, the accuracy of the tool has been measured in different physical target platforms and the results show this accuracy is near to 60%, and much of the error rate is due to the annotation process. The intermediate representation of LLVM is similar to assembler code but depending on the target architecture the difference can be significant due to this is a coarse-grain estimation.

V. CONCLUSIONS AND FUTURE WORK

Current embedded applications require fast and flexible performance analysis frameworks. These frameworks should be capable of supporting parallel applications and taking advantage of full computational capacity of the host.

The approach proposed in this paper enables host-compiled performance analysis for embedded parallel applications in current many-cores platforms. The gain obtained from scalability is slightly affected by the simulated platform. However, for this kind of simulation techniques the performance is limited by the number of synchronization and the threads execution time between synchronizations.

The experiments were performed on a multi-core workstation and the results obtained show the impact of simulation performance on parallel applications. The speed up of parallel simulations is similar to original code with the exception of particular cases.

In future works, an alternative in the kernel will be studied in order to reduce the penalty produced by the kernel when the code to execute by each thread is smaller than kernel computing. In addition, the code instrumentation process will be refined to achieve more accurate results. For such purpose, the estimation will be done using LLVM backend instead of the IR. This estimation will provide fine-grained annotation on the target architecture.

ACKNOWLEDGMENT

This work has been financed by the Mineco through the TEC2014-58036-C4-3-R project and CA112 HARP project; the Spanish MITyC and the EU through the Artemis 295371 CRAFTERS project.

REFERENCES

[1] E. Wandeler et all, "System architecture evaluation using modular performance analysis: a case study", Journal of Software Tools for Technology Transfer (STTT). Oct 2006.

[2] D. Yun, S. Kim, "A parallel simulation technique for multicore embedded systems and its performance analysis", IEEE Trans. on Computer-Aided Design of Integrated Circuit and Systems. Vol 31, No 1, Jan 2012.

[3] Y. Jung, J. Park, M. Petracca, L. Carloni, "NetShip: a networked virtual platform for large-scale heterogeneous distributed embedded systems". Proc. of Design Automation Conference, 2013.

[4] A. Gerslauer, "Host-compiled simulation of multi-core platforms", IEEE Int. Symp. on Rapid System Prototyping (RSP). June. 2011..

[5] P. Razaghi, A. Gerstlauer, "Predictive OS modeling for host-compiled simulation of periodic real-time task sets". IEEE Embedded Systems Letters. Vol 4, No 1. March 2012.

[6] S. Roloff, F. HAnnig, J. Teich, "Approximate time functional simulation of resource-aware programming concepts for heterogeneous MPSoCs". Proc. of Asian and South Pacific Design Automation Conference. ASP-DAC'12. 2012.

[7] C. Roth et all, "Asynchronous parallel MPSoC simulation on the single-chip cloud computer". IEEE Int. Symp. on System on Chip (SoC). 2012.

[8] S. Jafer, Q. Liu, G. Wainer, "Synchronization methods in parallel and distributed discrete-event simulation", Simulation Modeling Practice and Theory, 30 (2013).

[9] The LLVM Compiler Infrastructure: www.llvm.org

[10] Princeton Application Repository for Shared-Memory Computers (PARSEC) website: http://parsec.cs.princeton.edu

[11] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen Weihua Zhang and B. Zang, "COREMU: a scalable and portable parallel full-system emulator". ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2011). February, 2011.

[12] D.Strom and W. Gruener. "Pat gelsinger: A shot at running intel". Tom's Hardware Geuide, May, 2005.

[13] N. J. Gunther, "A Simple Capacity Model of Massively Parallel Transaction Systems", CMG National Conference, 1993.