sarteco

V Jornadas de Computación Empotrada

17-19 de septiembre de 2014, Valladolid

intel FUJITSU Departamento de Informática Universidad de Valladolid ETSI Informática Escuela Técnica Superior de Ingeniería Informática UVa

Actas de las V Jornadas de Computación Empotrada (JCE 2014)

Editor: Jesús González Peñalver
(c) 2014, V Jornadas de Computación Empotrada

ISBN-10: 84-697-0972-0
ISBN-13: 978-84-697-0972-6

Valladolid, 2014

ISBN 978-84-697-0972-6

9 788469 709726

VIPPE: Native simulation and performance analysis framework for multi-processing embedded systems

L. Diaz, E. Gonzalez, E. Villar, P. Sanchez

University of Cantabria
ETSIIT, Av. Los Castros s/n, 39005 Santander, Spain
{luisds, eduardo, villar, sanchez}@teisa.unican.es

Abstract—**Verifying the correctness of multi-processing embedded systems is a complex task. In order to avoid the cost, effort and time that the direct design verification on a physical prototype implies, simulation on a virtual model of the system is the most popular method used currently. Commercially available simulators require the complete SW binaries to be executed on each processing node. Although they can provide very accurate results, they can only be applied once the SW development has been completed. Native simulation technologies have been proposed to generate virtual platforms at the beginning of the design process, reducing porting efforts. As with any Discrete-Event simulation technique, native simulation presents problems in order to take advantage of the multi-processing capabilities of current host workstations where the simulation will be executed. Several concurrent simulated threads can be run in parallel in the host but to ensure deterministic behavior, it is necessary to synchronize all of them periodically in order to maintain causality among events. As a consequence, the number of cores that can be active during simulation is dramatically reduced. Embedded SW requires specific simulation techniques to take advantage of the multi-processing capabilities of workstations and to efficiently parallelize the simulation. In this paper, the results of the research effort towards an efficient, accurate-enough, parallel implementation of native simulation is presented.**

Keywords—**Native simulation, parallel SW simulation, performance analysis, embedded systems.**

I. Introduction

Nowadays, Embedded Systems (ESs) are designed and implemented using Multi-Processing, Systems-on-Chip (MPSoCs). Verifying the correctness of the design on these multiprocessing platforms is a complex task. In order to avoid the cost, effort and time that direct design verification on a physical prototype involves, simulation on a virtual model of the system is currently the most popular method used. Moreover, detecting design mistakes at the end of the design process may imply costly and time-consuming redesigns, compromising the final ES cost and time-to-market. As most of the functionality of the MPSoC is carried out in SW running on the different processing cores of the chip, efficient, accurate-enough SW simulation is becoming increasingly important.

Several SW simulation technologies at different abstraction levels have been proposed, providing different trade-off between accuracy and speed. Commercial simulation technologies are based on instruction set simulators (ISSs) and binary translation. However, none of them really provides the required trade-off for early evaluation. ISSs are usually very accurate but too slow to execute the thousands of simulations required to evaluate complete SoC design spaces. Simulations based on binary translation are commonly faster than ISSs but still too slow for design-space exploration. Additionally, in both cases, the simulation requires a completely developed SW and HW platform including the complete SW stack and operating systems in each computing node, fully operational peripheral models, libraries, device drivers, bus models, etc. Therefore they can only be applied once the development process is almost finished. Evaluating different allocations in heterogeneous platforms, different kinds of processors, different operating systems and SW optimizations is limited by the refining effort required to simulate all the options. Similarly, the evaluation of the effect of reusing legacy code in these infrastructures is not an easy task. As a consequence, faster and more flexible simulation techniques, capable of modeling the effect of all the components that impact on system performance are required for initial system development and performance evaluation.

Native simulation has proven to be a powerful simulation technology for early evaluation of different design alternatives. In native simulation, the application source code is compiled, the binary analyzed and the performance figures back-annotated to the source code which is compiled and simulated in the design workstation. Nevertheless, as with any Discrete-Event simulation technique, native simulation presents problems when trying to take advantage of the multi-processing capabilities of current host workstations where the simulation will be executed. Several concurrent simulated threads can be run in parallel in the host but to ensure deterministic behavior, it is necessary to synchronize all of them periodically to maintain causality among events. As a consequence, the number of cores that can be active during simulation is dramatically reduced.

Embedded SW requires specific simulation techniques in order to take advantage of the multi-processing capabilities of workstations and to efficiently parallelize the simulation. This paper presents the results of the research effort towards an efficient, accurate-enough, parallel implementation of native simulation.

II. State of the art

The dominant SW simulation technologies, such as ISSs [1-2] or virtualization (e.g. QEMU) [3-4], are based on models of the target processors executing the cross-compiled binary on the host. They require the availability of the complete SW stacks to be executed by each processing node including the HW-dependent-SW (HdS) and the OS. Although virtualizations achieve higher simulation speed than traditional ISS, both are associated with large simulation times when providing accurate execution times and power consumption estimations. As a consequence, reduction of design time and effort requires minimizing the number of simulation runs at this level of abstraction, thus performing architectural mapping decisions at a higher level. Nevertheless, virtual platforms based on binary simulation are the only method able to provide enough accuracy to ensure the functional and non-functional correctness of the design.

As an alternative, source-level models can provide enough accuracy with short execution times for design-space exploration. By instrumenting the code with back-annotated performance figures from an ISS, the accuracy can be increased significantly [7]. Two different techniques for timing annotation have been proposed. In trace-based simulation, the code is analyzed and commands inserted at certain points. A trace is a sequence of commands indicating the activity of the CPU executing the code. From this activity, the execution time and power consumption can be derived. As the trace is decoupled from a specific CPU, the technique may support different architectural mappings, scheduling policies and platform configurations. The traces are generated once and re-scheduled depending on the changes in design being analyzed such as different application mappings or task scheduling policies. When an abstract model of the OS is used, additional traces have to be considered [7]. Re-scheduling is avoided when a deterministic Model of Computation is used [5-6]. Trace-based simulation has been proposed as an alternative to virtualization in order to construct accurate virtual platforms for complex, heterogeneous many-core systems supporting DSE. Multiple atomic traces per basic block that allow an accurate reconstruction of the processor's behavior have to be used to achieve this goal[8]. Higher accuracy and flexibility in the architectural mapping alternatives on a heterogeneous platform come at the cost of a more complex analysis of a larger number of traces.

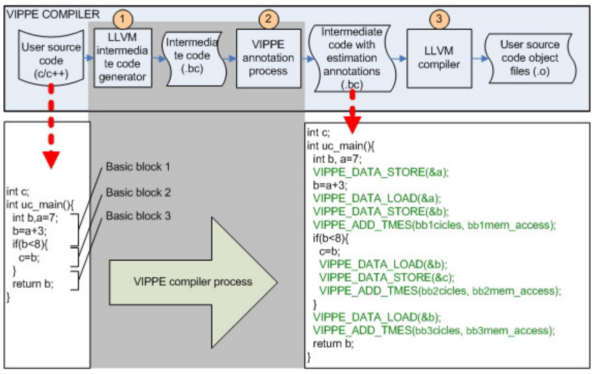
Native simulation technologies have been proposed for generating virtual platforms at the beginning of the design process, reducing porting efforts [9-12]. The methodology is very similar to trace-based simulation as an executable model of the system is used; in most cases, the complete application SW. The fundamental difference compared with trace-based simulation is that the code is instrumented directly with back-annotated information able to directly provide the estimated performance figures of execution times and power consumption. In this way, no additional analysis of the simulation results (traces) is needed. The performance estimation and code annotation can be done directly from the source-code or from the ISS after cross-compilation [10-12].

During recent years, there has been a growing interest in parallel implementations of host-compiled simulators. The synchronization strategy of these simulators can be synchronous [13] (if there are global synchronization points or clocks) or asynchronous [14]. The latter strategy improves simulation performance although the implementation is more complex. Another approach used to reduce simulation time is to relax the causality property (the cause must be before the effect). A conservative simulator guarantees that there are no causality errors while the optimistic approach [15] could have some errors that force the simulator to recover a previous state.

This paper presents a conservative parallel host-compiled simulator with an asynchronous synchronization strategy. Although other approaches synchronize the concurrent threads every time that a shared element is read or written, the proposed technique only requires synchronization during read operations. Additionally, it models the target RTOS and the NUMA structure of the target platform.

III. VIPPE

In native simulation, the embedded software is instrumented with some additional code during compilation. This new code provides several estimations (e.g. execution time and number of memory accesses) during the execution of the instrumented embedded-software code in the host platform.



```
int c;
int uc_main(){
  int b,a=7;
  b=a+3;
  if(b<8){
    c=b;
  }
  return b;
}
```

```
int c;
int uc_main(){
  int b, a=7;
  VIPPE_DATA_STORE(&a);
  b=a+3;
  VIPPE_DATA_LOAD(&a);
  VIPPE_DATA_STORE(&b);
  VIPPE_ADD_TMES(bb1cicles, bb1mem_access);
  if(b<8){
    c=b;
    VIPPE_DATA_LOAD(&b);
    VIPPE_DATA_STORE(&c);
    VIPPE_ADD_TMES(bb2cicles, bb2mem_access);
  }
  VIPPE_DATA_LOAD(&b);
  VIPPE_ADD_TMES(bb3cicles, bb3mem_access);
  return b;
}
```

Fig. 1. Example of code instrumentation for VIPPE.

Instrumented code is executed in the host platform with simulation libraries. In this execution, time, and consumption estimations are obtained.

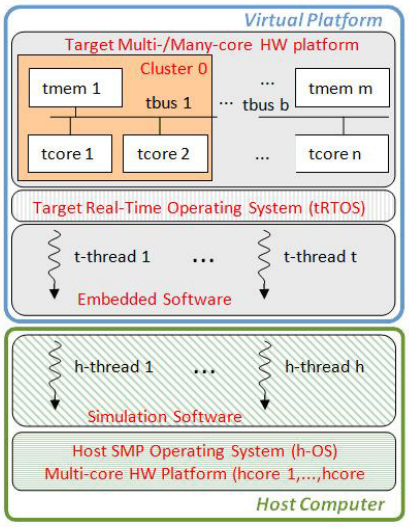


Fig. 2. Performance analysis framework.

Implementation of the virtual platform allocates every target thread to a host thread. Additionally, the target RTOS is implemented by an additional thread of the host (simulation kernel). For schedule target threads to target cores, the target RTOS implementation does not need to lock host threads. The threads are only locked when a shared variable or synchronization element (for example, a semaphore) has to be read (read synchronization). This approach minimizes locking and improves parallel execution.

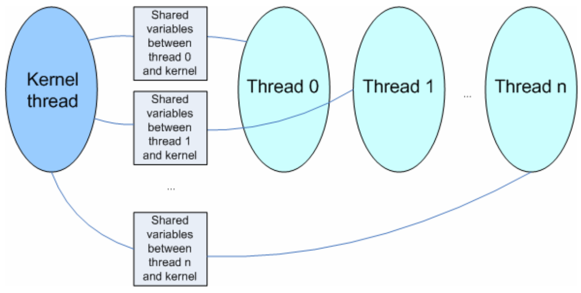

Kernel thread
Shared variables between thread 0 and kernel
Shared variables between thread 1 and kernel
...
Shared variables between thread n and kernel
Thread 0
Thread 1
...
Thread n

Fig. 3. Simulator structure.

The target simulated threads are responsible for modeling the functionality of the SW and informing the kernel thread about their estimations of individual thread times, access to memory, etc. With this information the kernel thread is responsible for modeling the system. This info is shared between the simulation kernel thread and the target simulated threads using shared variables.
The simulation kernel handles the physical simulation time or global time (globalSimTime).
This native simulation methodology has been implemented in the VIPPE (VIrtual Parallel platform for Performance Analysis) framework, a parallel version of the XXX (omitted for blind review) environment.

IV. VIPPE API

The VIPPE API consists of an optimal reduced set of primitives that allows us to make use of the services offered by the kernel.
These primitives constitute a metamodel that enables the implementation of higher level operative system interfaces. Therefore, this metamodel can be reused in order to implement different APIs such as POSIX, arduino, etc.

TABLE I

SET OF FUNCTIONS THAT FORM THIS API SORTED BY CATEGORIES

MANAGEMENT OF PROCESSES	MANAGEMENT OF TIMES
process_create	time_real_watch time_user_watch
MANAGEMENT OF THREADS	MANAGEMENT OF SEMAPHORES
get_prio set_prio get_id thread_create thread_exit thread_delete thread_wait_for_end	semaphore_create semaphore_wait semaphore_post semaphore_watch
MANAGEMENT OF SIGNALS	MANAGEMENT OF AFFINITIES
vippe_signal vippe_kill	uc_PE_mapping uc_PE_dismapp uc_take_OS_id

Function definitions:

int process_create(int prio): This function creates a new process. Receives the priority prio for the new process as the argument and returns an execution thread identifier. Upon unsuccessful completion, the function returns a negative value.
int get_prio(): This function returns the priority of the calling thread.
int set_prio(int prio): This function sets prio as the calling thread priority.
int get_id(): This function returns the calling thread identifier.
int thread_create(int prio, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg): This function creates a new thread with priority prio and attributes attr that executes the function start_routine. The arguments are passed through the pointer arg.
int thread_delete(int thread_id): This function finishes the execution of the thread with identifier thread_id.
int thread_wait_for_end(int thread_id): This function suspends execution of the calling thread until the thread with identifier thread_id finishes its execution.
int semaphore_create(int init_value): This function creates a semaphore with value init_value and returns its identifier.
int semaphore_wait(int sem, long long int time_out): This function locks the semaphore with identifier sem by performing a semaphore lock operation.

-If time_out is equal to zero:

If the semaphore value is zero the calling thread will not return from the call until it either locks the semaphore or the call is interrupted by a signal.

-Otherwise:

If the semaphore value is zero the calling thread will not return from the call until it locks the semaphore, the call is interrupted by a signal or the amount of time indicated by time_out has expired.

int semaphore_post(int sem): This function unlocks the semaphore with identifier sem by performing a semaphore unlock operation on that semaphore.
int semaphore_watch(int sem): This function returns the value of the semaphore with identifier sem.
long long int time_real_watch(): This function returns the execution time of the system.
long long int time_user_watch(): This function returns the execution time of the user thread.

int vippe_signal(int signal_id, void (*function_ptr)()): This function establishes the function function_ptr as the handler for the signal signal_id.
int vippe_kill(int signal_id, int thread_id): This function sends the signal signal_id to the thread with identifier thread_id.
int uc_PE_mapping(int id_PE): The calling thread is migrated to the process element with identifier id_PE.
int uc_PE_dismapp(): This function unbinds the calling thread from any processor previously bound by calling the function uc_ PE_mapping.
int uc_take_OS_id(): This function returns the identifier of the OS to which the calling thread belongs. This function is necessary for affinity management since it must be executed on the processors controlled by the OS to which they belong.
As is shown later, an implementation of POSIX has been developed from this API, proving this set to be enough.
This API provides independency between the kernel and user program sides. As an advantage, if a windows system is to be simulated, it is not necessary to modify the simulator kernel: but only to implement a WinAPI from VIPPE API functions in the same way it has been done for POSIX.
On the other hand, if the simulator is to be executed on a different native OS (i.e. windows), it is possible to reuse the POSIX API, it being necessary only to implement the kernel simulator for that specific OS.

V. POSIX API

POSIX has been chosen as the operative system interface API, since it is one of the most widely used.
Our set of POSIX API functions has been divided into the categories of events, concurrency, synchronization, timing and I/O, depending on the service they provide.
Events: Signals and interruptions related functions. **Concurrency:** Execution thread management. Process and thread creation, cancellation, etc.
Synchronization: Inter-process/thread synchronization. Semaphores, mutex, etc.
Timing: Obtaining time related functions.
I/O: Involves data input/output functions, for example writing/reading files.

TABLA II

BASIC VIPPE API FUNCTIONS INVOLVED IN IMPLEMENTATION

Events	vippe_signal vippe_kill
Concurrency	process_create thread_create thread_delete thread_wait_for_end get_id get_prio set_prio
Synchronization	semaphore_create semaphore_wait semaphore_post semaphore_watch
Timing	time_real_watch time_user_watch
I/O	semaphore_wait semaphore_post

The previous table shows the basic VIPPE API functions involved in the implementation of the functions of each category, however, there are no rigid boundaries among the different categories, it being necessary for some POSIX functions to use additional VIPPE API functions not included in the category. An example is the implementation of the POSIX function sem_timedwait() that makes use of time_real_watch() for time calculations in addition to semaphore_wait().
The POSIX API implementation only makes use of the elements of synchronization provided by the VIPPE API which means that accesses to POSIX's own critical section data (such as, for example, POSIX's internal list of current threads) is controlled by these elements, not by the native operative system.
Using native OS synchronization methods would lead to invalid simulation time values, since waiting times resulting from semaphore blocks would not be added to the simulation time.
VIPPE API simplifies POSIX portability to a new target platform by implementing the reduce set of VIPPE API functions in assembly code.
For example, if there is a new platform available, but there is still no OS ported for that platform (e.g. Linux), the assembly implementation of VIPPE API (that requires less effort and cost than implementing Linux), allows the user program to be executed since the POSIX API implementation is based on the VIPPE API.
Not all POSIX functions need to be implemented making use of the VIPPE API.
Functions that do not require synchronizations that may generate blocks, making use of signals, obtaining time information or providing execution thread management do not need to be implemented making use of VIPPE API provided that these functions do not directly modify or use elements created from VIPPE API functions.
For example, function pthread_sigmask(int how, const sigset_t *set, sigset_t *oset) changes the signal mask of the current thread and needs to be implemented. However, function sigempyset(sigset_t *set) that initializes the signal set pointed to by *set* does not need to be implemented.

A. *POSIX API annotation*

In order to obtain accurate simulated time values, times added by POSIX must be taken into account.
Three different approaches are proposed in order to obtain these values:

1. Fixed execution time as a parameter of the function.
An execution time is estimated for each function. This value is passed as an argument to the overloaded function as follows:

```
posix_function(...,execution_time) {
        increment_simulated_time(execution_time);
        ...
        ...
}
```

Therefore every simulated execution time of the same function takes exactly the same time (except for time increments due to possible blocks).

2. Annotating each possible path inside the function, so simulated execution time of the function will depend on the path taken through the function.

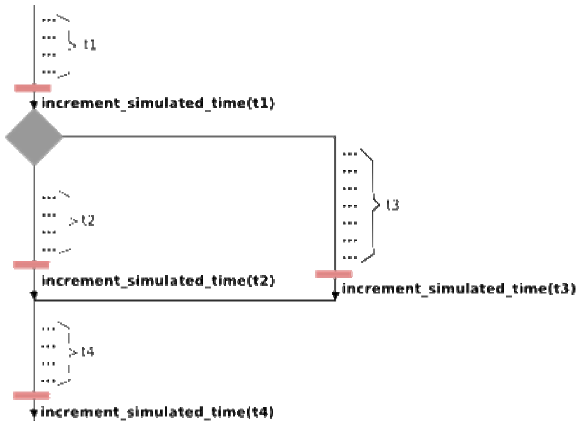

t1
increment_simulated_time(t1)
t2
increment_simulated_time(t2)
t3
increment_simulated_time(t3)
t4
increment_simulated_time(t4)

Fig. 4. Annotation for each possible execution path.

This approach is more accurate than the previous one since functions do not always execute in the same way, resulting in different execution times.

3. Parsing the POSIX API code.
By this method the functions comprising the POSIX API are parsed in the same way used for the user program. Results in simulation execution times are similar to the previous approach but this also has an extra advantage. Using this method it is unnecessary to include the functions for simulation time increments inside the POSIX functions, this allows us to execute the user program using POSIX on a platform with an available VIPPE API implementation as explained previously.
Since the second and third methods provide more accurate results and the third one provides advantages compared to the second one, parsing the POSIX API code is chosen as the annotation method.

While high precision is achieved for user program simulation times, obtaining errors around 10%, less accurate results are obtained for POSIX function simulation times. This is due to the fact that the function execution times depend on POSIX implementation, which varies according to the OS implementation on the real platform, whereas simulation times are given by our POSIX implementation.
However, OS weight is expected to be small in comparison to the total execution time, making this error less significant.

VI. Experimental results

The VIPPE framework includes the VIPPE API implementation, and the proposed implementation in this paper of POSIX API uses VIPPE API. This enables the embedded application to be executed without code manipulation.
The performance of the proposed methodology has been evaluated with the PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark suite [17]. PARSEC integrates several multithreaded programs in which the user can define the number of threads that the application uses. This is very useful for evaluating the relation between number of target threads and parallel simulation performance.
The evaluation is focused on the execution time of the host-compiled parallel simulation. There is no information about the accuracy of the proposed simulation because this parameter mainly depends on the source-code timing annotations and they are independent of the native simulation methodology (the paper's main objective).
The benchmark and the original code have been executed in a host with 8 Intel Xeon E5-2687W at 3.10GHz. Every processor has 8 cores, thus the host platform integrates 64 cores with SMP capability. The computer has 64Gb of RAM and 20Mb of cache.
Figure 5 analyzes the relationship between the speed-up and the number of target threads. The number of cores of the target platform has been limited to 4. The host platform uses 32 cores. Although the native simulation requires more host time than the original benchmark, the speed-up is similar. The original (native) sequential code is 64.7 times faster than the host-compiled simulation but the maximum difference between the two speed-ups is about 15%. This demonstrates the limited impact of the proposed methodology and the advantages of allocating target threads to host threads (the simulation has a similar speed-up to the original description when the number of target threads is modified).

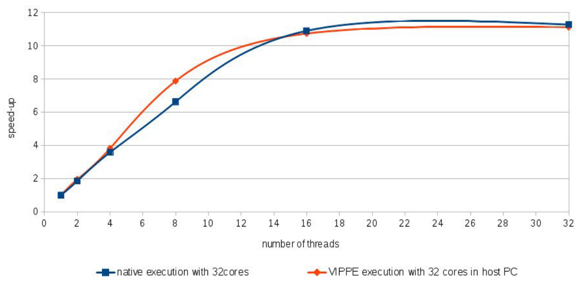


Fig. 5. Speed-up with the number of target threads

Fig 6 analyzes the relationship between the execution time and the number of cores in the target platform. This figure demonstrates that performance improvement in simulation execution time is independent (or at most weakly dependent) on the number of processors in the target platform and only on the number of host processors and application threads. The measurements shown in figure 6 have been obtained executing a x264 example of the PARSEC benchmark suite (H.264 video encoding) with 4 concurrent threads and 1920x1080 with 25fps input video.

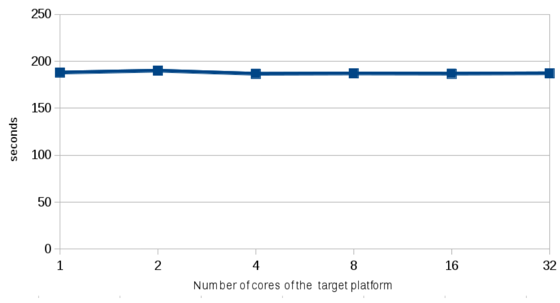


Fig. 6. Time vs number of cores in the target platform

Fig 7 analyzes the relationship between the execution time and the number of cores in the host computer and the number of threads in the simulated application.

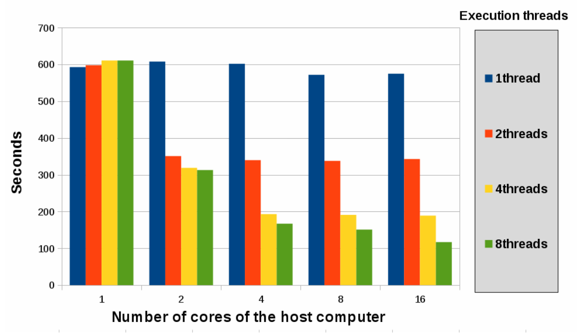


Fig. 7 Execution time vs number of cores in the host computer vs number of threads in the simulated application

When the number of threads is higher than the number of host processors, the time penalty due to thread creation, scheduling time (i.e. context switching), etc. increases execution time. This behavior is highlighted in figure 7 with one host core (with two threads, execution time is higher than with one, and with three higher than with two etc.).

When the number threads is equal to or lower than the number of host processors, it can be seen that the execution time of the simulation is reduced proportionally to the number of threads in the application.

VII. CONCLUSIONS

In this paper the architecture of an efficient, parallel, native simulation tool has been described. The tool supports the simulation of application SW running on any multi-processing platform providing an OS API on an abstract model of the RTOS and the processing HW. An optimized, general-purpose API has been developed. Although simple, it has proven to support more complex OS services such as POSIX. Experimental results show the advantages provided by the tool in simulating the application SW on multi-core workstations.

ACKNOWLEDGMENT

This work has been financed by the Mineco through the TEC2011-28666-C04-02 project and CA112 HARP project; the Spanish MITyC and the EU through the Artemis 332913 CopCams project and Artemis 295371 CRAFTERS project.

REFERENCES

[1] L. Benini, et al: "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC", Journal of Signal Processing Systems, 2005.
[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] M.-C. Chiang, T.-C. Yeh and G.-F. Tseng: "A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, V.30, N.4, April 2011.
[4] 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.
[5] M. Streubühr, R. Rosales, R. Hasholzner, C. Haubelt and J. Teich: "ESL Power and Performance Estimation for heterogeneous MPSoCs using SystemC", FDL Conference, 2011.
[6] A.D. Pimentel: "The Artemis workbench for system-level performance evaluation of embedded systems", Int. J. Embedded Systems, V.3, N.3, 2008.
[7] R. Plyaskin, A. Masrur, M. Geier, S. Chakraborty and A. Herkersdorf: "High-level timing analysis of concurrent applications on MPSoC platforms using memory-aware trace-driven simulations", Int. Conference on VLSI and System-on-Chip, IEEE, 2010.
[8] R. Leupers, G. Martin, R. Plyaskin, A. Herkersdorf, F. Schirrmeister, T. Kogel and M. Vaupel: "Virtual Platforms: Breaking new grounds", DATE Conference, 2012.
[9] J. Schnerr, O. Bringmann, A. Viehl, W. Rosenstiel: "High-performance timing simulation of embedded software". DAC conference, 2008.
[10] H.Shen, M-M. Hamayun and F. Pétrot: "Native Simulation of MPSoC using Hardware-Assisted Virtualization", IEEE Trans. on Computer-Aided design of Integrated Circuits and Systems, V.31, N.7, July, 2012.
[11] H. Posadas, S. Real, E. Villar: "M3-SCoPE: Performance Modeling of Multi-Processor Embedded Systems for Fast Design Space Exploration", in C. Silvano, W. Fornaciari & E. Villar (Eds.): "Multi-objective Design Space Exploration of Multiprocessor SoC Architectures: the MULTICUBE Approach", Springer, 2011.
[12] S. Chakravarty, Z. Zhao and A. Gerstlauer: "Automated, retargetable back-annotation for host compiled performance and power modeling", proc. Of 2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), ACM, 2013.
[13] 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.
[14] C. Roth et all; "Asynchronous Parallel MPSoC Simulation on the Single-chip Cloud Computer". IEEE Int. Symp. on System on Chip (SoC). 2012.
[15] S. Jafer, Q. Liu, G. Wainer; "Synchronization methods in parallel and distributed discrete-event simulation", Simulation Modeling Practice and Theory, 30 (2013).
[16] www.llvm.org
[17] http://parsec.cs.princeton.edu