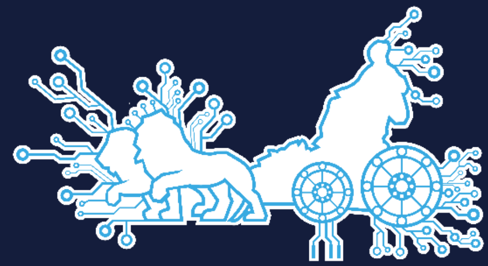
XXIX Conference on
Design of Circuits and Integrated Systems

DCIS 2014



November 26th – 28th, 2014
Madrid, Spain

VIPPE, parallel simulation and performance analysis of multi-core embedded systems on multi-core platforms.

L. Diaz, E. Gonzalez, E. Villar, P. Sanchez
TEISA, University of Cantabria
Santander, Spain
{luisds, eduardo, villar, sanchez}@teisa.unican.es

Abstract — **Verifying the correctness of multi-processing embedded systems is a complex task and in addition to that system-on-Chips (SoC) are integrating a continuously growing number of cores. 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 causes 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, however, ensuring deterministic behavior requires synchronizing 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. This paper proposes a native simulation framework, called VIPPE, which makes an efficient use of the multi-core host platform. The approach has been evaluated with a benchmark of the PARSEC suite and the results show that the simulation speed-up (with the number of target threads) is close to the original application speed up. This demonstrates the limited impact on performances of the proposed simulation parallelization methodology.**

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 the direct design verification on a physical prototype implies, simulation on a virtual model of the system is the most popular method used currently. Moreover, detecting design mistakes at the end of the design process may imply costly and time consuming redesign, compromising the final ES cost and time-to-market. As most of the functionality of the MPSoC is provided by SW running on the different processing cores of the chip, efficient, accurate-enough SW simulation is becoming increasingly important.

The dominant SW simulation technologies, such as ISSs [1] or virtualization (e.g. QEMU) [2], 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 virtualization achieves higher simulation speed than traditional ISS, both are associated with large simulation times when providing accurate execution time and power consumption figures. 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 way 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 [5]. Two different techniques for timing annotation have been proposed. In trace-based simulation, the code is analysed 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 [5]. Re-scheduling is avoided when a deterministic Model of Computation is used [3-4]. 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. To achieve this goal, multiple atomic traces have to be used per basic block, allowing an accurate reconstruction of the processor's behavior [6]. Higher accuracy and flexibility in the architectural mapping alternatives on a heterogeneous platform come at the cost of a more complex analysis of a higher number of traces.

Native simulation technologies have been proposed to generate virtual platforms at the beginning of the design process, reducing porting efforts [5-10]. 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 provide directly the estimated performance figures of execution times and power consumption. In this way, no additional analysis of the

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.

978-1-4799-5743-9/14/$31.00 ©2014 IEEE.

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 [8-10].

As with any Discrete-Event simulation technique, native simulation has problems taking 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 behaviour 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 concurrently during simulation is dramatically reduced.

Embedded SW requires specific simulation techniques in order to take advantage of the multi-processing capabilities of current workstations and efficiently parallelize the simulation.

In this paper, the results of the research effort towards an efficient, accurate-enough, parallel implementation of native simulation are presented. Specifically, in this paper VIPPE is introduced. VIPPE is a native simulator that enables a fast and high-level estimation of the software execution time in a virtual target platform. The proposed native simulator technique makes efficient use of the multi-core host platform. In order to execute user code with VIPPE, a POSIX API has been implemented to enable the communication between the user code and the simulation kernel, which is explained in section III. Finally in section IV, experimental results are shown in order to evaluate the efficient implementation of the simulator.

II. Proposed Native Simulator Technique

This paper presents a conservative parallel native 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 architecture of the target platform.

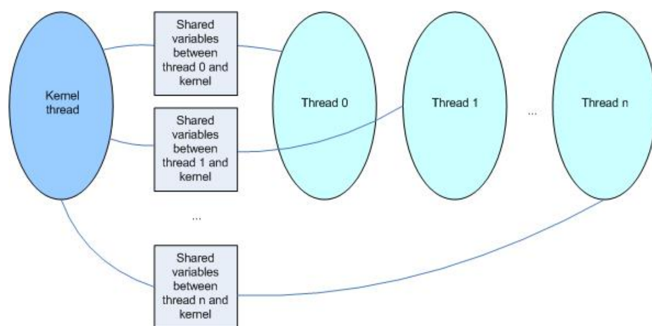


Fig. 1. Simulator structure.

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). To schedule target threads to target cores, the target RTOS implementation does not require host threads to be locked. They 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.

Target simulated threads are responsible for modeling the functionality of the SW and informing the kernel thread about their estimations of individual threads' times, access to memory, etc. With this information, the kernel thread models the system. This information is shared between the simulation kernel thread and target simulated threads using shared variables.

The description of the host-compiled methodology is divided into four sections. The first section describes the code instrumentation used, the second explains the variables used for the kernel needed in the simulation, the third details the time assignment methodology, and the last section describes the simulation kernel algorithm.

A. *Code instrumentation*

In a host-compiled 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. This work uses the llvm compiler framework[11] to analyze and annotate the embedded source code. The instrumentation code computes several parameters such as:

- **status(i).** Thread "i" has two possible states: active (it is being executed) and locked (it is waiting to read a shared variable or it is waiting until an event occurs, for example post to semaphore).
- **localTime(i).** This is the total execution time of the instrumented code in the target thread "i". It is the local execution time of the thread and it only includes the execution time of the target core instructions. Thus, this time does not include memory access time.
- **memAcc(i,k).** This is the total number of accesses from thread "i" to memory "k". This value could take into account cache behavior. As cache modeling does not affect the paper's conclusions, this work does not include cache simulation although it is implemented in the current version of the tool.
- **localEventList(i).** The list includes all the write-accesses from thread "i" to shared variables and synchronization elements. It is sorted by thread local time (localTime). Every element in the list includes three values:
    - The shared variable or synchronization element in which the thread "i" writes a value.
    - The new value or action (for example, post to semaphore).
    - The thread local time at which the value is written. The list is sorted in increasing order of local thread time.

B. *Simulation kernel variables*

The simulation kernel uses a lock-free method, that is based on atomic operation, to read several thread-"i" parameters: status(i), localTime(i) and memAcc(i,k). They are copied in equivalent kernel variables: **kStatus(i)**, **kLocalTime(i)** and

kMemAcc(i,k). Additionally, the kernel defines several variables:

- **schedLocalTime(i)**. This is the local time of thread "i" that has been already executed (or scheduled) in the target platform cores. As this work uses a conservative simulation approach, the scheduled time is less than or equal to the local time of the thread ("**schedLocalTime(i)** <= **kLocalTime(i)**"). The difference "**kLocalTime(i)-schedLocalTime(i)**" is the time that has to be scheduled during the next simulation cycles.
- **schedMemAcc(i,k)**. This is the number of memory accesses from thread "i" to memory "k" that has already been executed. Thus, the difference "**kMemAcc(i,k)-schedMemAcc(i,k)**" is the number of memory accesses that has to be allocated (or scheduled) during the next simulation cycles.
- **SimStatus(i)**. In the simulation kernel, this is the status of thread "i". The methodology defines three possible states: "init", "active" and "locked". When a new thread is created, its simulation state is "init" until the "new-thread creation event" is scheduled. At this point, the thread status changes to "active" or "locked" in the kernel. A thread has an "active" status when it has local time to schedule, thus "**kLocalTime(i)-schedLocalTime(i) > 0**". A thread has "locked" state if it does not have time to schedule ("**kLocalTime(i)-schedLocalTime(i) = 0**") and **kStatus(i)=**"locked". In this case, the thread does not have local time to schedule and it is waiting for an event or shared-variable value.

The simulation kernel 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.

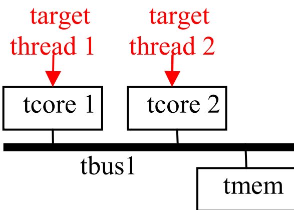


Fig. 2. Target platform architecture and thread allocation.

C. *Time assignment methodology*

This section presents a simple example of thread scheduling in a target platform. The target architecture is shown in Figure 2. The platform includes a bus and a memory, although zero-delay bus and memory access is assumed in this sub-section. The target RTOS model allocates every thread to a target core (see Figure 2).

The yellow column in Figure 3 shows the values of the thread local variables and simulation kernel parameters at simulation time 5000 units. A time slice is the period of time for which a thread (or process) is allowed to run uninterrupted in a target RTOS with pre-emptive scheduling. In this work, it is assumed that a time-slice period will increase the simulation time for a simulation cycle. In Figure 3, the blue column shows the values of the simulation-related variables after a time slice. A time slice consumes 100 time units in Figure 3.

First, the simulation kernel reads the thread local times (localTime) and updates the "klocalTime" values (atomic read operation). This allows the host to execute other target threads in parallel with the simulation kernel. For example, thread 1 has executed 1000 time units during the kernel execution in Figure 3 (localTime(1) = 1000 + 1000). We can observe that both threads have been executed for more than a time-slice period: "localTime(i) -schedLocalTime(i) >= 100" (for thread 1, "1000-600=400 > 100" and for thread 2 "500-400=100"). Thus, both threads can be scheduled in this simulation cycle. When the thread is scheduled, the "schedLocalTime" is increased by a time-slice period (see blue-column values). Finally, the shared variable can be updated. In this example, the notation "a=1 @750" means that variable "a" will take value "1" at time 750. Taking into account that the thread "schedLocalTime" parameter is synchronized with the "globalSimTime" when a thread is being scheduled, it is possible to map the local events of the thread list to current values (see variable "b" in the blue column). The localEvent List of thread 2 has an event (b=2) at local time 450 (yellow column). When the kernel schedules thread 2, the schedLocalTime of thread 2 (400) is equivalent to the global simulation time (5000). This means that all events that occur at local time 400, occur at global simulation time 5000. Thus, the event "b=2" that occurs at local time 450 will be allocated at global simulation time 5050.

In Figure 3, the red color is used to highlight the values that have been modified during the simulation cycle. It is also important to comment that this time assignment methodology is lock-free.

Local Thread variables

Target thread	i=1	i=2
status(i)	active	locked
localTime(i)	1000	500
localEventList	a=1 @750	b=2 @450

i=1	i=2
active	locked
2000	500
a=1 @750	

Simulation kernel variables

Target thread	i=1	i=2
SimStatus(i)	active	active
klocalTime(i)	...	...
schedLocalTime(i)	600	400

i=1	i=2
active	locked
1000	500
700	500

globalSimTime	***5000***
Current value of a	a=3 @ 4000
Current value of b	b=1 @3000

5100
a=3 @ 4000
b=2 @ 5050

Fig. 3. Thread scheduling with a time slice of 100 units.

D. *VIPPE simulation algorithm*

Figure 4 presents the algorithm that schedules target threads in the target platform during a time slice. This algorithm is executed in a host thread that is concurrently executed with the target threads. The algorithm has three parts. First, a set of threads will be allocated in target cores. They will be executed during a time slice period (Step 2 in Figure

5). Second, the threads that are locked by a synchronization element will check the current value of these elements. They will resume their execution when it is possible (Step 3). Third, the last test (Step 4) determines that the simulation has finished.
Step 2 is the main loop of the algorithm and it begins with an update of the thread local times. Step 2.2 implements the scheduling policy of the target RTOS and selects a set of threads that will be scheduled during Step 2's execution. Step 2.3 checks that all the selected threads have enough time to be schedule.

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 are 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.

Fig. 4. Simulation kernel algorithm.

III. APLICATION INTERFACE

Application executions make use of different OS interfaces depending on the available OS (i.e., an application executed over linux uses POSIX API).
Communication between this OS interface and the VIPPE kernel is established through a proprietary kernel API (vippe API).
Therefore, the VIPPE API consists of an optimal reduced set of primitives that allows the operating system APIs to make use of the services offered by the kernel.
These primitives constitute a metamodel that allows higher level operating system interfaces to be implemented. Hence, this metamodel can be reused in order to implement different OS APIs as POSIX, Arduino, WINAPI, etc.
This layered model of communication is depicted in the next figure.
An advantage is that if we wish to use the simulator on a different native OS (i.e. windows), it is possible to reuse the POSIX API, it is only necessary to implement the kernel simulator for that specific OS.
In this framework an implementation of POSIX has been carried out from this VIPPE API, proving this set to be enough.
POSIX has been chosen as the operating system interface API, since it is one of the most widely used.

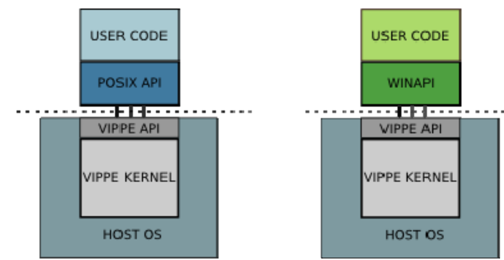


Fig. 5. The figure shows POSIX and WINAPI implementations from VIPPE API, respectively.

A. VIPPE API

VIPPE API functions are classified in different categories according to the functionality they cover as shown in table I.
Management of process functions allows the creation of new processes.
Management of thread functions allows creation and deletion of threads and for other thread finalizations to be awaited. It can also be used for setting thread priority and obtaining thread id and priority.
Management of time functions allows to read the execution simulated times while simulation. The simulated execution time of the system and simulated execution time of the user thread can be obtained.
Management of semaphore functions involve functios related with synchronization among the different processes. These functions provide mechanisms for kernel semaphore creation, increment (post) and decrement (wait) operations as well as enabling the current simulation semaphore value to be obtained.
Management of signal functions provides a mechanism for thread synchronization through signals. Hence, these functions allow signal delivery and signal handling function selection.
Management of affinity functions enables threads to be bound and unbound to/from specific processor elements as well as allowing information to be obtained about which OS each thread belongs to (necessary information since threads must be executed on the processors controlled by the OS to which they belong).

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_delete thread_wait_for_end	semaphore_create semaphore_wait semaphore_post semaphore_watch
MANAGEMENT OF SIGNALS	MANAGEMENT OF AFFINITIES

<table>
<tr><td>vippe_signal
vippe_kill</td><td>uc_PE_mapping
uc_PE_dismapp
uc_take_OS_id</td></tr>
</table>

B. POSIX API

Our set of POSIX API functions has been divided into the categories of events, concurrency, synchronization, timing and I/O depending on the service provided.

Events: Involves signal, interruptions and other functions that provoke a change in the execution flow.

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.

Table II 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 their 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().

TABLE II

BASIC VIPPE API FUNCTION INVOLVED IN IMPLEMENTATION

<table>
<tr><td>EVENTS</td><td>TIMING</td></tr>
<tr><td>vippe_signal
vippe_kill</td><td>time_real_watch
time_user_watch</td></tr>
<tr><td>CONCURRENCY</td><td>SYNCRONIZATION</td></tr>
<tr><td rowspan="3">process_create
thread_create
thread_delete
thread_wait_for_end
get_id
get_prio
set_prio</td><td>semaphore_create
semaphore_wait
semaphore_post
semaphore_watch</td></tr>
<tr><td>I/O</td></tr>
<tr><td>semaphore_wait
semaphore_post</td></tr>
</table>

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 (as for example POSIX's internal list of current threads) is controlled by these elements, not by the native operating system.

Using native OS synchronization methods would result in invalid simulation time values since waiting time resulting from semaphore block would be ignored and would not be added to the simulation time.

The fact that the POSIX API is implemented just making use of the services offered through the VIPPE API provides an extra advantage: implementing these VIPPE API functions in assembly code for a specific platform is all that is needed to execute the user program on that platform, it not being necessary to port an OS.

If for example there is a new platform available, but there is still no OS (i.e Linux) ported for that platform, the assembly implementation of VIPPE API that requires less effort and cost than implementing Linux allows the user program to be executed given that the POSIX API implementation needs no changes due to the fact that it is based on the VIPPE API.

In order to obtain accurate simulated time values, times added by POSIX must be taken into account.

The POSIX API is parsed in the same way used for the user program. Using this method, it is unnecessary to include the functions for simulation time increment within 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.

While high precision is achieved for user program simulation times obtaining errors around 10% , less accurate results are obtained for POSIX functions 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.

IV. EXPERIMENTAL RESULTS

The VIPPE framework includes theVIPPE API implementation, and the proposed implementation in this paper of POSIX API uses VIPPE API. This enables the execution of embedded applications without code manipulation.

The performance of the proposed methodology has been evaluated with the PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark suite [12]. 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 6 analyzes the relation 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 similar a speed-up to the original description when the number of target threads is modified).

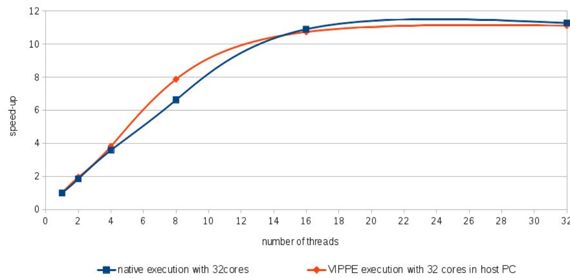


Fig. 6. Speed-up with the number of target threads

Fig 7 analyzes the relation between the execution time and the number of cores in the target platform. This figure demonstrates that the performance improvement in simulation execution time is independent (or at least weakly dependent) of the number of processors in the target platform and depends only on the number of host processors and application threads. The metrics shown in figure 7 have been obtained executing a x264 example of the PARSEC benchmark suite (H.264 video encoding) with 4 concurrent threads and a 1920x1080 with 25fps of input video.

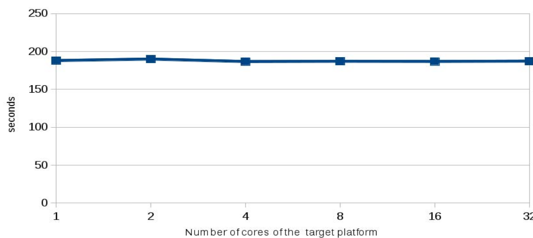


Fig. 7. Time vs number of cores in the target platform

The next experiment evaluates the impact of the number of host cores on the simulation performance. In Linux, it is possible to define the number of cores that an application uses (set processor affinity). Using this capability, we have limited the number of host cores to several values (1, 2, 4, 8 and 16 host cores). The host-compiled simulator increases the number of target threads by one because it inserts a new thread: the simulation kernel. This affects simulation performance: it is not possible to obtain a speed-up of x16 with 16-target threads (17 host threads).

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 shown in figure 8 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.

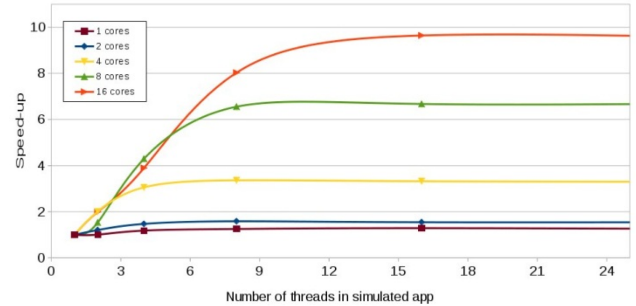


Fig 8. Speed-up with the number of host cores

V. Conclusions

In this paper the architecture of an efficient, parallel, native simulation tool has been described. The tool supports the simulation of the application SW running on any multi-processing platform providing a 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.

References

[1] L. Benini, et al: "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC", Journal of Signal Processing Systems, 2005.

[2] 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.

[3] M. Streubühr, R. Rosales, R. Hasholzner, C. Haubelt and J. Teich: "ESL Power and Performance Estimation for heterogeneous MPSoCs using SystemC", FDL Conference, 2011.

[4] A.D. Pimentel: "The Artemis workbench for system-level performance evaluation of embedded systems", Int. J. Embedded Systems, V.3, N.3, 2008.

[5] 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.

[6] R. Leupers, G. Martin, R. Plyaskin, A. Herkersdorf, F. Schirrmeister, T. Kogel and M. Vaupel: "Virtual Platforms: Breaking new grounds", DATE Conference, 2012.

[7] J. Schnerr, O. Bringmann, A. Viehl, W. Rosenstiel: "High-performance timing simulation of embedded software". DAC conference, 2008.

[8] 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.

[9] 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.

[10] 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.

[11] www.llvm.org

[12] http://parsec.cs.princeton.edu