Concurrency Emulation and Analysis of Parallel Applications for Multi-Processor System-on-Chip Co-Design

Giovanni Beltrame
European Space Agency
Keplerlaan 1
Noordwijk, The Netherlands
giovanni.beltrame@esa.int

Luca Fossati, Donatella Sciuto
Politecnico di Milano
Piazza Leonardo da Vinci
Milano, Italy
{fossati, sciuto}@elet.polimi.it

ABSTRACT

This paper presents a novel technique for the modeling and the simulation of parallel applications for Multi-Processor Systems-on-Chip (MPSoCs). This technique consists of an application-transparent emulation of OS primitives, including task creation, scheduling, synchronization etc.; this emulation guarantees compatibility with any program compiled against the standard POSIX library, independently of the target OS. This methodology can be used to perform initial HW/SW partitioning and concurrent engineering of a given application, as it allows any software routine to be transparently emulated with SystemC modules. The proposed approach has been verified on a large set of multi-threaded benchmarks, with both POSIX Threads and OpenMP programming styles. Results show that our methodology enables (a) fast simulation of POSIX applications, (b) accurate analysis of multi-threaded applications, and (c) co-design and fast preliminary hardware-software partitioning.

This work has been partially supported by the European HiPEAC network of excellence.

Categories and Subject Descriptors

I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*Model Validation and Analysis*

General Terms

Design, Measurement, Performance

Keywords

OpenMP, MPSoC, codesign, Operating System, Emulation, ReSP

1. INTRODUCTION

Industry migration from implicit to explicit concurrency in processor architectures is facing many issues, for which there are few solutions [8]. As new programming paradigms and compilers are debated research topics, the application designer has to manually identify, capture, schedule and map concurrency. Moreover, increasingly large portions of electronic systems are being implemented in software and its development cost starts dominating the cost for the whole system. Software is also becoming the critical part of the development schedule, mainly because deploying and testing it on the real target hardware is complicated. The concurrent design of hardware and software aims at mitigating these issues, allowing software to be developed before the final hardware is ready. Three main methodologies have been devised [15]: the use of an FPGA emulator, the use of a workstation with appropriate scaffolding, or running the software in a virtual platform (full system simulator). In particular, the latter technique offers an approach that has many advantages (flexibility, accuracy, observability, etc.) with respect to the other methods.

When developing parallel programs, the identification of performance bottlenecks is an increasingly important and challenging task, due to the complex relationship between application, operating system, and underlying hardware configuration. In order to gather timing details as soon as possible in the development process we need high level models of such intricate relationships. Accurate simulation also requires taking into account the effect of the Operating System. Unfortunately, most of the available high level co-simulation methodologies suffer from the so called *code equivalence* problem (as presented in [15]) in that the code executed by the virtual system is different from the code executed by the real hardware. This not only affects the application timing, but it may also change the overall system behavior leading to less-than-optimal or wrong design choices.

In this work we address the aforementioned issues by providing a co-design environment suitable for the development of multi-processor systems. The base idea consists in a transparent emulation of OS primitives: this emulation guarantees full compatibility with any program compiled against the POSIX library (including OpenMP and Pthread based applications); it can also be easily ported to other embedded software standards. The presented methodology can be applied to a variety of tasks, such as evaluation of the performance of OpenMP constructs, the measurement of the inherent concurrency of the software under development, and the identification of optimal OS configurations. In addition, we present how it is possible to apply these ideas for initial design space exploration.

This paper is organized as follows: Section 2 describes previous research on the subject and Section 3 presents how the proposed methodology addresses the identified issues. Finally, Section 4 shows the experimental results and Section 5 draws some concluding remarks.

2. PREVIOUS WORK

The HW/SW Co-design flow [14] usually starts at system-level, when the boundaries between the hardware and software parts of the final system have not yet been established. After functional verification, the HW/SW partitioning takes place and co-simulation is employed to validate and refine the system. The tight market constraints and the high complexity of current designs push to anticipate as early as possible thisHW/SW co-validation phase, even before an Instruction Set Simulator (ISS) for the target architecture is available. These motivations, together with the fact that simulation speed of an ISS is usually low, have pushed for the addition of OS models in system-level Hardware Description Languages (HDL) (as SystemC or SpecC). This allows native execution of both the hardware and software models of the system, thus consistently speeding-up simulation. In addition, since both hardware and software partitions are described using the same HDL, it is easy to move functionalities between them.

The authors of [15] present a technique to automatically generate timed OS simulation models; these models partially re-use the high level OS code, thus mitigating the code equivalence problem. Since execution is native on the host machine, the timing of the target architecture is not accurately replicated. On the contrary, we use ISSs in our approach: this means lower simulation speed but also, as the assembly code of the final application is used, a reduction of the code equivalence problem. We also do not depend on a particular OS, so a wider design space exploration is possible.

Schirner and Domer [12] address the problem of modeling preemption and interrupts in abstract RTOS models; they apply the Result Oriented Modeling (ROM) technique in order to guarantee accurate interrupt response time while maintaining fast simulation speed. They mainly concentrate on simulating the system timing behavior and the code equivalence problem is not taken into account.

An untimed abstract model of an RTOS is presented in [6]; the model supports all the services of the $\mu ITRON$ standard, therefore it can be used with a wide range of applications, but this work is applied only to uniprocessor systems.

A different approach, valid during the early phases of the design cycle, is used in [2]; the authors automate the translation of SystemC processes into threads of the underlying OS. This mechanism enables quick evaluation of the possible assignments of tasks to hardware or software components. The same objectives are reached in [9] by mapping OS thread management primitives into SystemC ones. Due to limitations in the SystemC process model, these works do not achieve true concurrency. Girods et al. [5] also address the lack of support for embedded software development by working in the .NET environment.

These approaches are valid to help the designer perform and refine the HW/SW partition, but they do not help in the validation of the system functionalities (for the code equivalence problem), and they are very limited in the assessment of the system's timing properties. On the other hand, the execution on an ISS of the exact same software which will be deployed on the embedded system creates problems because the OS has to be already chosen and ported to the target hardware. Moreover, having the OS ready means it might be difficult to refine the HW/SW partitioning since the OS should be updated accordingly.

In this paper we propose a way to emulate OS primitives in order to minimize *code equivalence* issues while still maintaining *independence from specific OSes*. Our technique also enables early HW/SW codesign, and it allows the evaluation of different synchronization strategies for multiprocessor systems.

3. PROPOSED METHODOLOGY

For the implementation and the evaluation of the design methodology presented in this paper we use the Open Source simulation platform ReSP [1].

ReSP is based on the SystemC library and it targets the modeling of Multi-Processor Systems. Its most peculiar feature consists in the integration of C++ and Python programming languages; this augments the platform with the concept of reflection [4], allowing full observability and control of every C++ or SystemC element (variable, method, etc.) specified in any component model.

In this work, we exploit and extend ReSP's characteristics to include a System Call Emulation subsystem. This is used for the preliminary exploration of the applications' bottlenecks, for guiding the designer in the choice of the target OS and as a support for early HW/SW codesign. The rest of this Section is devoted to a more thorough explanation of such concepts.

3.1 System Call Emulation

System Call Emulation is a technique which allows the execution of application programs on an Instruction Set Simulator (ISS) without the need to simulate a complete OS. The low level calls made by the application to the OS routines (*system calls*, SC) are identified and intercepted by the ISS, and then redirected to the host environment which takes care of their actual execution. Suppose, for example, that an application program contains a call to the `open` routine to open file *"filename"*. Such a call is identified by the ISS using the mechanisms described below and routed to the host OS, which actually opens *"filename"* on the PC's filesystem. The file handle is then passed back to the simulated environment.

As outlined in Section 1, having a simulation framework with System Call Emulation capabilities allows the application developers to start working as early as possible, even before a definite choice about the target OS is performed. These capabilities are also used for ISS validation, by enabling fast benchmark execution. Thus, it is common to find emulation features in instruction set simulators. *Sim-It ARM* [10] and *SWARM* [13] simulate the Service Call (`svc`) instruction of the ARM processor so that the corresponding request is forwarded to the host OS. This requires the emulation mechanism to have knowledge of the various `svc` codes used by the different compilers. ArchC [11], on the other hand, modifies to the compiler in order to insert jumps to predefined addresses instead of the SC code; the ISS then takes the appropriate action when one of these particular addresses gets associated to the program counter.

Figure 1 shows an overview of our System Call emulation mechanism. Each ISS communicates with one centralized

Trap Emulator (TE): this is the component responsible for forwarding the System Calls from the simulated to the host environment. In order to ensure independence between the ISS and the TE, interfaces (IF) are created, and communication between the TE and the ISS exclusively takes place through IFs. In our approach, instead of identifying the SCs

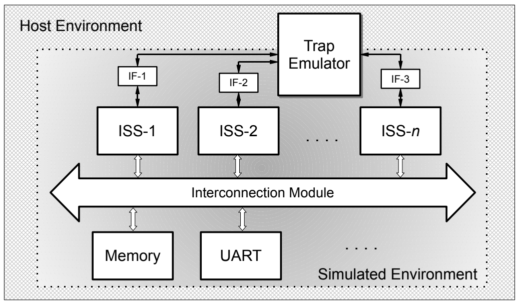


Figure 1: Organization of the simulated environment including the System Call Emulation module

through ad-hoc assembly instructions or special addresses, the name (i.e. the *symbol*) of the routine corresponding to the SC is used. When the application program is loaded for execution, the names of the low level SCs (e.g. `sbrk`, `_open`, etc.) are associated with their addresses in the application's binary file. At runtime the ISS checks for those addresses and, when one matches the current program counter, the corresponding functionality of the TE is called and the SC is emulated. Figure 3 contains a more detailed explanation of the mechanism for the `sbrk` SC: as the library function *malloc* executes, it calls `sbrk`; the TE checks if the current program counter is to be trapped, and as it recognizes `sbrk`, it stops the ISS execution and performs the appropriate actions.

With respect to previous work, our implementation shows the following advantages:

1. *Independence from the cross-compiler toolchain*: since the names of the system call routines are used, there is no need, for example, to adhere to the `svc` convention with which the software is built or to create fictitious jumps in the code
2. *High interoperability with different ISS types*: the IF is the only component which needs customization to allow a new ISS to be integrated with the TE
3. *Extensibility*: the presented mechanism can also be used for preliminary hardware/software partitioning. Moreover, by emulating the POSIX-Threads routines, a multi-processor, multi-threading concurrency manager was implemented

Since only the low level SCs (e.g. `sbrk`) are emulated and the rest of the OS code (e.g. `malloc`) is executed unmodified in the ISS, our method maintains high code equivalence with the final software, even at the assembly level.

The remainder of the section is devoted to present in detail all the components of the emulation mechanism and how they can be used both for HW/SW codesign and for the emulation and management of concurrency in a single- or multi-processor environment.

3.1.1 *Processor Interface*

Communication between the *Emulator* and the *Instruction Set Simulator* is a critical point in the overall design: on one hand it has to be designed so that ISSs can be easily plugged into the system and, on the other hand, it has to be as fast as possible to guarantee high simulation speed. These requirements are in contrast with each other, thus the right tradeoff has to be determined. As shown in Figure 2, two solutions were identified: the first one is purely based on the C++ language and it does not exploit any of the *reflective* features of ReSP; the second one uses the Python language to unintrusively access the ISS internal variables.

Writing pure C++ code sacrifices ease of use and flexibility for speed, meaning that the ISS code has to be slightly modified in order to pass, at every cycle, the program counter to the IF which, in turn, checks if it corresponds to the address of one of the emulated routines. The biggest drawback of this technique is that it requires access to the source code of the ISS, which might not be always available.

The second solution is based on the *reflective* properties of ReSP. Instead of having the ISS push each program counter change to the TE, the TE reads it when necessary. This is performed by using *callbacks*, methods which are called when events happen in the simulator. In this case the triggering event is a change in the program counter of each ISS. With Python reflective capabilities and its integration with the C++ language and the SystemC library, callbacks can monitor the ISS without the need for any modification to its source code. Unfortunately, this does not come for free: the callback mechanisms is partly implemented in C++ and partly in Python, and the switching between them is subject to some overhead and consequent simulation slowdown (see Section 4).

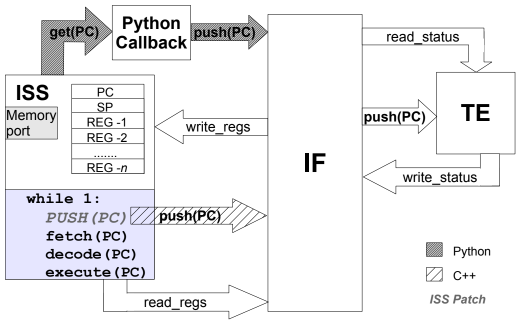


Figure 2: Communication mechanisms between the ISS and the IF; the C++ or Python parts are alternative solutions

With respect to the communication from the TE to the ISS, both solutions use the same mechanism: the IF exports a set of methods which allow to (a) access the registers (PC, SP, etc.), (b) access the memory as seen by the processor, and (c) manage the Application Binary Interface (ABI) of the target architecture. It is worth noting that the IF must have knowledge of how the ISS is implemented in order to work correctly. This means that new IFs should be created for the correct integration of new ISS models.

3.1.2 *Concurrency Manager*

The majority of the embedded systems used nowadays is composed of more than one processing unit, either in heterogeneous or homogeneous configuration. To fully exploit those systems, parallel applications are needed. Such applications require proper functionalities to manage task creation and termination and to manage synchronization among them. While these functionalities are normally provided by an OS, the number of approaches based on dedicated hardware components is growing, especially in the embedded world.

The TE was extended for the emulation of concurrency management routines with an additional unit, called *Concurrency Manager*. The overall mechanism is analogous to the one depicted in Figure 3, but instead of trapping I/O or memory management, the TE traps routines for thread creation, destruction, synchronization etc. For this purpose, we created a placeholder library containing all the symbols (i.e. the function identifiers) of the POSIX-Thread standard, with no implementation present. This ensures that programs using the `pthread` library can correctly compile. During execution, all calls to `pthread` routines are trapped and forwarded to the Concurrency Manager.

If the application software is compiled with a recent GNU GCC compiler (at least version 4.2) it is also possible to successfully simulate OpenMP directives. In addition, by correctly trapping the `libgomp` routines, it is possible to directly emulate OpenMP directives, hence enabling a quick evaluation of their implementation using dedicated hardware components. More details on this are presented in Section 3.2.

Many embedded systems operate in real-time environments and/or are composed of real-time subsystems. While the current implementation of the Concurrency Manager is not targeted to such systems, it can easily be extended to support them by modifying the scheduling policy, by introducing the concept of task priorities, and by enabling the modeling of interrupts.

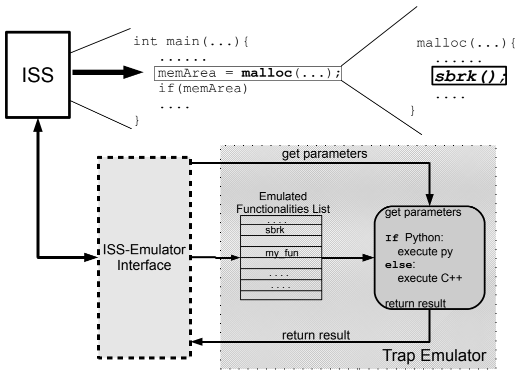


Figure 3: Function Trap Emulator (TE)

3.2 Extension for HW/SW Codesign

While all the features introduced by the TE can be successfully employed for software design and verification, their applicability extends further. As shown in Figure 3, the extensions implemented in ReSP allow the designer to "trap" any routine, not just the ones belonging to the Operating System or to the POSIX-Thread library. This enables an effective and fast evaluation of the different possible implementations of a given functionality f:

a) execution of the unchanged *software routine*: no trap is installed for f and the functionality of f is simulated on the ISS

b) *emulation*: a trap is installed for f and its behavior emulated; since it is possible to associate a custom delay d to the trap handler, we can explore how the timing of the system would be if f were implemented with a hardware module of delay d

c) implementation through a *hardware module*: the trap handler triggers the execution of a SystemC defined hardware module containing the functionality of f

Also in this situation, Python can be used to describe the trap handler and/or the SystemC module for f, thus increasing productivity and programming ease.

Since it is possible to associate a custom delay d with any trap handler, the described techniques can be used to explore the system behavior in front of threading and synchronization primitives with different latencies, helping in the choice of the operating system and, for example, in deciding whether hardware managed threading is needed.

4. EXPERIMENTAL RESULTS

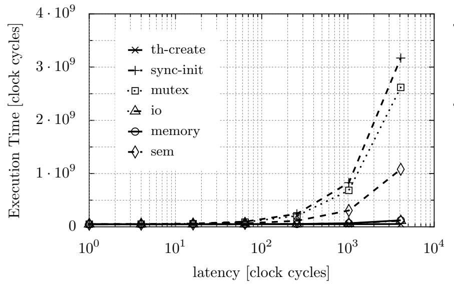
This Section presents the results obtained by applying the described techniques to real world applications and to benchmark suites. In particular, we show how the trapping and emulation capabilities of the system can be used (a) for hardware/software codesign, (b) for the estimation of the parallelism embedded in a given software application and (c) for the estimation of thread management overhead. Finally we present an analysis on how the introduced techniques impact over simulation speed.

All tests have been executed using ReSP on a multi-ARM architecture consisting of a variable number of cores with caches, a shared memory, and timers, interrupt controllers and UARTs as needed by the OS (for the experiments where we used it), all interconnected by a shared bus, as shown by Figure 1. The processors are clocked at 500MHz, and the bus at 250MHz. Simulations where timing was recorded were run on a Core 2 Duo 2.66GHz Linux machine.

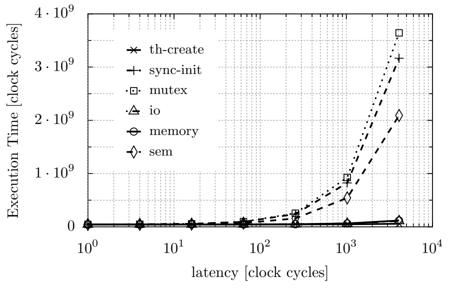
4.1 OS Emulation for Co-Design

To prove the usefulness of the methodology for co-design, we analyzed the impact of OS latencies on a set of parallel benchmarks. For this purpose, 8 benchmarks chosen from the OpenMP Source Code Repository [3] were used. All OS primitives were divided into 6 classes: thread creation (*th-create*); synchronization initialization, e.g. mutex, semaphore, and condition variable creation (*sync-init*); mutex locking and unlocking (*mutex*); semaphore waiting and posting (*sem*); memory management (*memory*); and general I/O (*io*). Since it is possible to associate custom latencies to emulated routines, we can observe how different values (corresponding to different system configurations) affect the system.

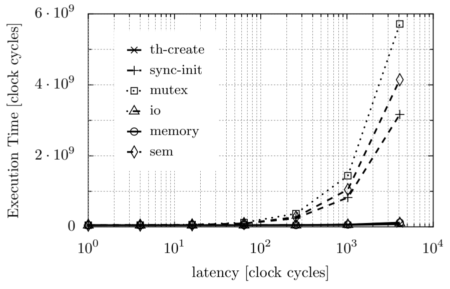
Figure 4(e) shows the average behaviour of the 8 benchmarks when the number of cores, as well as the number of OpenMP threads, ranges from 2 to 16 in powers of 2. On each of these hardware configurations, the benchmarks were run with exponentially increasing latencies for each class, yielding a total of 1344 simulations. The trend is that, for increasing latency, synchronization primitives are the ones that affect execution time most, while I/O and memory management have negligible effect. As a more specific example,



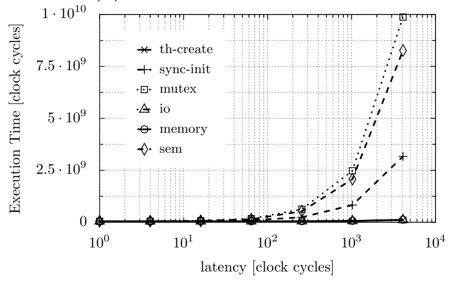
(a) 2 PE 2 threads



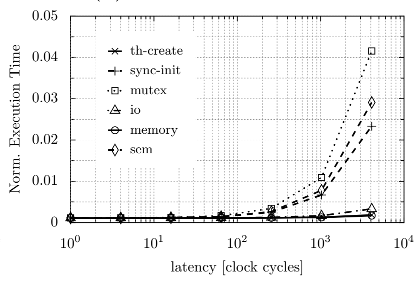
(b) 4 PE 4 threads



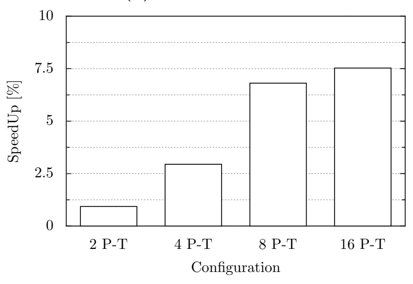
(c) 8 PE 8 threads



(d) 16 PE 16 threads



(e) Global Average



(f) Hardware Synchronization

Figure 4: (a)-(d): execution Time in front of different System Call latencies; (e) shows the average execution time on all the 1344 runs for the different 8 benchmarks ; (f) shows the increase in performance when using a hardware synchronization module in different configurations

Figure 4 shows this effect for the `lu` benchmark: the impact of synchronization grows noticeably and quickly exceeds initialization costs as the number of processors and threads is increased. This kind of analysis can guide the designer on the decision about the OS architecture, also taking into account hardware-supported OS mechanisms.

In fact, targeting the latencies to an actual operating system (e.g. eCos) can give a quick estimation of the performance of an application with a given HW/OS configuration. Figure 4(f) shows `lu`'s performance when a hardware accelerator for managing synchronization primitives is inserted in the system. Assuming a memory-mapped module with a 4-cycle latency (considering bus contention), we observe an increasing speedup with the number of processors, saturating around 8 processors (mainly due to the lack of parallelism of the benchmark). For these experiments the average latencies of the *eCos* OS for each of the 6 primitive classes were measured. The runs were, then, performed by setting the TE latencies to these values.

In general, as for synchronization primitives, any software function can be trapped and emulated in either SystemC or Python. This means that the proposed methodology can be used to quickly shift functionalities from software to hardware, without modifying the application's source code, thus helping the designer in the initial partitioning.

4.2 Overhead Estimation

The proposed methodology can also be used to estimate the inherent parallelism of an application and the overhead introduced by thread management. We examined two multi-threaded open source applications and an OpenMP benchmark: *ffmpeg*, a powerful audio and video encoder/decoder, *pbzip2*, a parallel block compression algorithm, and *CG*, a scientific application kernel from the NASA Parallel Benchmarks suite [7]. For these tests, a 4-core, 4-threaded architecture was used.

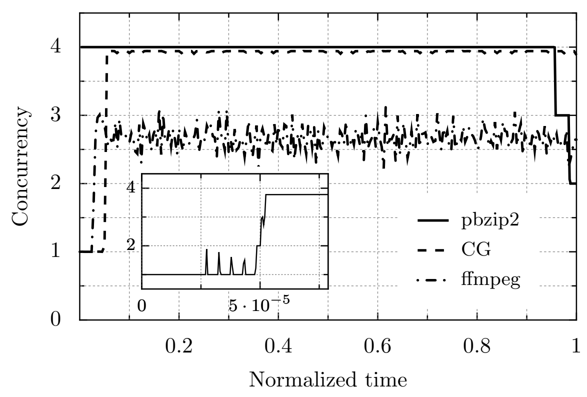


Figure 5: Concurrency profiles on a 4-core architecture; the initialization of pbzip2 is shown in detail

The main idea of these tests is to run the application setting all the concurrency-related latencies to zero, hence exposing the application's parallel behaviour. The intuition behind this is that, in the ideal case, when thread management costs 0, the speedup given by the execution on n threads is equal to n; in practice this does not happen, obtaining a speedup of $m \leq n$: $n - m$ is a measure of the application inherent parallelism. The concurrency profiles presented in Figure 5 were measured averaging the number of active processors for each clock cycle over 250 points, equally distributed along the time axis. To measure the overhead introduced by thread management, we compare the average concurrency expressed by each application with the actual speedup obtained against single-threaded execution as $overhead = (1 - speedup/concurrency) \times 100$.

The concurrency profile of `ffmpeg` transcoding (decoding and encoding) a short clip from MPEG1 to MPEG4 shows that, due to data dependencies and to the structure of ffmpeg's code, the 4 processors are never fully exploited, with an average 2.6 processors working at the same time. When compared to a single-core architecture, ffmpeg is 2.3 times faster, yielding a 11% overhead. pbzip2, instead, offers more exploitable concurrency as its algorithm has less data dependencies. After a very small initial setup time, pbzip2 fully exploits the available resources. However, even if the average concurrency expressed by pbzip2 is 3.4, its speedup on 4 cores is only 2.9, resulting in a 15% overhead. Finally, CG shows the benefits of OpenMP programming, with a very smooth concurrency profile, locked to 3.9 after the initialization phase, and averaging at 3.77. CG's speedup amounts to 3.55, yielding a small 9% overhead introduced by the compiler's OpenMP runtime (GNU libgomp in our case). All results are summarized in Table 1.

Table 1: Application runtimes and overheads

Name	Concurrency	Speedup	Overhead
ffmpeg	2.6	2.3	11%
pbzip2	3.4	2.9	15%
CG	3.8	3.5	9%

4.3 Performance

To evaluate the performance of the proposed methodology, 8 OmpSCR benchmarks were run on the eCos OS, using a 4-core virtual platform. eCos system calls were profiled on the 4-ARM architecture, and the average latency of each class of system calls was determined. The derived latencies were introduced for each system call in our OS emulation system, and the benchmarks were executed once more; this same procedure was adopted to create Figure 4(f). Figure 6 shows that even with this simple scheme, the methodology can very well emulate the behavior of a specific OS, with an average error of 9%. This is a reasonable error as full code equivalence is present for the application and library functions, but threading, multi-processor management, and low-level OS functions are emulated. We plan to further validate the accuracy of this approach by applying it to other OSs as RTEMS, Embedded Linux, etc.

In addition, the use of the OS emulation layer introduces a noticeable speedup when compared to running the OS on each ISS. This is due to several factors, including the absence of hardware components emulated by the TE, such as UARTs and Timers, and the fact that, in our mechanism, idle processors do not execute busy loops but they are, instead, suspended.

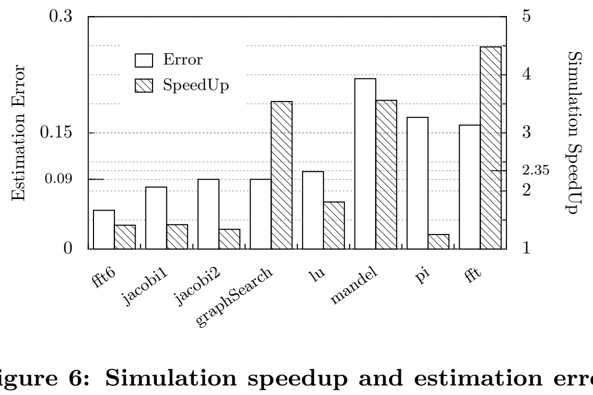


Figure 6: Simulation speedup and estimation error using the Emulation Layer instead of eCos

Finally, the overhead introduced when writing one of the emulated functions in Python was measured Table 2 shows that the use of Python slows down the function's execution time by 17% when compared to pure SystemC solution.

Table 2: Execution time for Python and C++ based trap handlers

Language	Avg. Time (sec.)	STD (sec.)
Python	1.35	0.09
C++	1.19	0.09
Slowdown $(C++/py)$	17%	

5. CONCLUDING REMARKS

In this paper we presented an innovative mechanism for Operating System emulation inside Instruction Set Simulators. In addition to being non intrusive in the ISS source code, the described techniques are extended for the emulation of POSIX-Threads routines and multi-processor concurrency management. High code equivalence is maintained, enabling fast and accurate simulation of POSIX applications.

Our methodology has been applied to many real world concurrent applications and benchmarks using both POSIX-Threads and OpenMP programming styles. Results show that emulation brings an average speedup of 2.35 with an average error of 9%. We also described how the proposed methodology was applied to the measurement of the inherent parallelism of software applications and how it can be used for early HW/SW codesign.

6. REFERENCES

[1] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto. ReSP: a non-intrusive transaction-level reflective MPSoC simulation platform for design space exploration. In *ASP-DAC '08*, pages 673–678, 2008.
[2] P. Destro, F. Fummi, and G. Pravadelli. A smooth refinement flow for co-designing HW and SW threads. In *DATE '07*, pages 105–110, 2007.
[3] A. J. Dorta. The openMP source code repository. In *PDP'05*, pages 244–250, Feb. 2005.
[4] B. Foote and R. E. Johnson. Reflective facilities in smalltalk-80. pages 327–335, 1989.
[5] B. Girodias, E. M. Aboulhamid, and G. Nicolescu. A Platform for Refinement of OS Services for Embedded Systems. In *DELTA '06*, pages 227–236, 2006.
[6] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada. RTOS-centric hardware/software cosimulator for embedded system design. pages 158–163, 2004.
[7] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance.
[8] E. A. Lee. The problem with threads. *Computer*, vol. 39:pag. 33–42, 2006.
[9] H. Posadas, J. Ádamez, P. Sánchez, E. Villar, and F. Blasco. POSIX modeling in SystemC. pages 485–490, 2006.
[10] W. Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *DATE '03*, page 10556, 2003.
[11] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: a SystemC-based architecture description language. In *SBAC-PAD 2004*, pages 66–73, 2004.
[12] G. Schirner and R. Domer. Introducing preemptive scheduling in abstract RTOS models using result oriented modeling. In *(DATE '08)*, 2008.
[13] SWARM http://www.cl.cam.ac.uk/~mwd24/phd/swarm.html.
[14] W. Wolf. A decade of hardware/software codesign. *Computer*, 36:38–43, 2003.
[15] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya. Automatic Generation Including Fast Timed Simulation Models of Operating Systems in Multiprocessor SoC Communication Design. 2002.