2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

# VirtualSoC: a Full-System Simulation Environment for Massively Parallel Heterogeneous System-on-Chip

Daniele Bortolotti, Christian Pinto, Andrea Marongiu, Martino Ruggiero and Luca Benini
*DEI - University of Bologna*
*Viale Risorgimento, 2 - 40136 Bologna, Italy*
*Email:[name].[surname]@unibo.it*

***Abstract*—Driven by flexibility, performance and cost constraints of demanding modern applications, heterogeneous System-on-Chip (SoC) is the dominant design paradigm in the embedded system computing domain. SoC architecture and heterogeneity clearly provide a wider power/performance scaling, combining high performance and power efficient general-purpose cores along with massively parallel many-core-based accelerators. Besides the complex hardware, generally these kinds of platforms host also an advanced software ecosystem, composed by an operating system, several communication protocol stacks, and various computational demanding user applications. The necessity to efficiently cope with the huge HW/SW design space provided by this scenario makes clearly full-system simulator one of the most important design tools. We present in this paper a new emulation framework, called VirtualSoC, targeting the full-system simulation of massively parallel heterogeneous SoCs.**

## I. Introduction

Performance modeling plays a critical role in the design, evaluation, and development of computing architecture of any segment, ranging from embedded to high performance processors. Simulation has historically been the primary vehicle to carry out performance modeling, since it allows for easily creating and testing new designs several months before a physical prototype exists. Performance modeling and analysis are now integral to the design flow of modern computing systems, as it provides many significant advantages: i) accelerates time-to-market, by allowing the development of software before the actual hardware exists; ii) reduces development costs and risks, by allowing for testing new technology earlier in the design process; iii) allows for exhaustive design space exploration, by evaluating hundreds of simultaneous simulations in parallel.

High-end embedded processor vendors have definitely embraced the heterogeneous architecture template for their designs as it represents the most flexible and efficient design paradigm in the embedded computing domain. Parallel architecture and heterogeneity clearly provide a wider power/performance scaling, combining high performance and power efficient general-purpose cores along with massively parallel many-core-based accelerators. Examples and results of this evolution are AMD Fusion [1], NVidia Tegra [2] and Qualcomm Snapdragon [3]. Besides the complex hardware, generally these kinds of platforms host also an advanced software eco-system, composed by an operating system, several communication protocol stacks, and various computational demanding user applications.

Unfortunately, as processor architectures get more heterogeneous and complex, it becomes more and more difficult to develop simulators that are both fast and accurate. Cycle-accurate simulation tools can reach an accuracy error below 1-2%, but they typically run at a few millions of instructions per hour. The necessity to efficiently cope with the huge HW/SW design space provided by this target architecture makes clearly full-system simulator one of the most important design tools. Clearly, the use of slow simulation techniques is challenging especially in the context of full-system simulation. In order to perform an affordable processor design space exploration or software development for the target platform, trade-off accuracy for speed is thus necessary by implementing new virtual platforms that allow for faster simulation speed at the expense of modeling fewer micro-architecture details of not-critical hardware components (like the host processor domain), while keeping high-level of accuracy for the most critical hardware components (like the manycore accelerator domain).

We present in this paper VirtualSoC, a new virtual platform prototyping framework targeting the full-system simulation of massively parallel heterogeneous system-on-chip composed by a general purpose processor (i.e. intended as platform coordinator and in charge of running an operating system) and a many-core hardware accelerator (i.e. used to speed-up the execution of computing intensive applications or parts of them). VirtualSoC exploits the speed and flexibility of QEMU, allowing the execution of a full-fledged Linux operating system, and the accuracy of a SystemC model for many-core-based accelerators.

The specific features of VirtualSoC are:

- Since it exploits QEMU for the host processor emulation, unmodified operating systems can be booted on VirtualSoC and the execution of unmodified ARM binaries of applications and existing libraries can be simulated on VirtualSoC.
- VirtualSoC enables accurate manycore-based accelerator simulation. We designed a full software stack allowing the programmer to exploit the hardware accelerator model implemented in SystemC, from within a user-space application running on top of QEMU. This software stack comprise a Linux device driver and a user-level programming API.
- The host processor (emulated by QEMU) and the SystemC accelerator model can run in an asynchronous way, where a non-blocking communication interface has been implemented enabling parallel execution between QEMU and SystemC environments.
- Beside the interface between QEMU and the SystemC model, we also implemented a synchronization protocol able to provide a good approximation of the global system time.
- VirtualSoC can be also used in stand-alone mode, where only the hardware accelerator is simulated, thus enabling accurate design space explorations.

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.177

IEEE computer society

To the best of our knowledge, we are not aware of any existing public domain, open source simulator that rivals the characteristics of VirtualSoC. This paper focuses on the implementation details of VirtualSoC and evaluates the performance of various benchmarks and presents some example case studies using VirtualSoC.

The rest of the paper is structured as follows: in Section II we provide an overview of related work, in Section III we present the target architecture, focusing on the many-core accelerator in Section IV. The implementation of the proposed platform is discussed in Section V. Software simulation support is described in Section VI, finally experimental results and conclusions are presented in Sections VII and VIII.

## II. Related work

The importance of full-system emulation is confirmed by the considerable amount of effort committed by both industry and research communities in developing such designing tools as more efficient as possible. We can cite several examples, like Bochs [4], Simics [5], Mambo [6], Parallel Embra [7], PTLsim [8], AMD SimNow [9], OVPSim [10] and SocLib [11].

QEMU [12] is one of the most widely used open-source emulation platform. QEMU supports cross-platform emulation and exploits binary translation for emulating the target system. Taking advantage of the benefits of binary translation, QEMU is very efficient and functionally correct, however it does not to provide any accurate information about hardware execution time. In [13] authors have implemented program instrumentation capabilities to QEMU for user application program analysis. This work has only been done for the user mode of QEMU and it cannot be exploited for system performance measurements (e.g. device driver). Moreover, profiling based on program instrumentation can heavily change the execution flow of the program itself, leading to behaviors which will never happen when executing the program in the native fashion. Authors in [14] have instead presented pQEMU, which simulates the timing of instruction executions and memory latencies. Instruction execution timings are simulated using instruction classification and weight coefficients, while memory latency is simulated using a set-associative cache and TLB simulator. This kind of approach can lead to a significant overhead due to the different simulation stages (i.e. cache simulation, TLB simulation), and even in this case the proposed framework can only run user-level applications without the support of an operating system.

QEMU lacks also of any accurate co-processors simulation capabilities. Authors in [15] interfaced QEMU with a many-core co-processor simulator running on an nVidia GPGPU [16]. Despite the co-processor simulator described in [16] is able to simulate thousands of computing units connected through a NoC, it runs at a high level of abstraction and does not provide precise measurements from the simulated architecture. Moreover authors do not address the problem of timing synchronization between QEMU and the co-processor simulation.

Other works have been mainly concentrated on enabling either cycle accurate instruction set simulators for the general purpose processor part or SystemC-based simple peripherals, without considering complex many-core-based accelerators [17].

When interfacing QEMU with the SystemC framework, several implementation aspects and decisions need to be accurately taken into account, since development choices can limit and constraint the performance of the overall emulation environment. The optimal implementation should not possibly affect efficiency, flexibility and scalability.

Establishing the communication between QEMU and SystemC simulator through inter-process communication socket is another approach. Authors in [18] use such facility between a new component of QEMU, named QEMU-SystemC Wrapper, and a modified version of the SystemC simulation kernel. The exchanged messages have the purpose not only to transmit data and interrupt signals but also to keep the simulation time synchronized between the simulation kernels. However using heavy processes does not allow fast and efficient memory sharing, which in this case can be achieved only using shared memory segments. Moreover, Unix Domain Sockets are less efficient, in terms of performance and flexibility, than direct communication between threads.

QEMU-SystemC [19] allows devices to be inserted into specific addresses of QEMU and communicates by means of the PCI/AMBA bus interface. However, QEMU-SystemC does not provide the accurate synchronization information that can be valuable to the hardware designers. [20] integrates QEMU with a SystemC-based simulation development environment, to provide a system-level development framework for high performance system accelerators. However, this approach is based on socket communication, which strongly limits its performance and flexibility. Authors in [21] suggested an approach based on threads since context switches between threads are generally much faster than between processes. However, communication among QEMU and SystemC uses a unidirectional FIFO, limiting the interaction between QEMU and the SystemC model.

We present in this paper a new emulation framework based on QEMU and SystemC which overcomes these issues. We chose QEMU amongst all simulators cited (e.g. OVPSim [10], Soclib [10]) because it is fast, open-source and also very flexible enabling its extension with a moderate effort. Our approach is based on thread parallelization and memory sharing to obtain a complete heterogeneous SoC emulation platform. In our implementation the target processor and the SystemC model can run in an asynchronous way, where non-blocking communication is implemented through the use of shared memory between threads. Beside the interface between QEMU and a SystemC model, we also present a lightweight implementation of a synchronization protocol able to provide a good approximation of a global system time. Moreover, we designed a full SW stack allowing the programmer to exploit the HW model implemented in SystemC, from within a user-space application running on top of QEMU. This software stack comprise a Linux device driver and a user-level programming API.

## III. Target Architecture

Modern embedded SoCs are moving toward systems composed by a general purpose multi-core processor accompanied by a more energy efficient and powerful many-core accelerator (e.g. GPU). In these kinds of systems the general purpose processor is intended as a coordinator and is in charge of running an operating system, while the many-core accelerator is used to speed up the execution of computing intensive applications or parts of them. Despite their great computing power, accelerators are not able to run an operating system due to the lack of all needed surrounding devices and to the simplicity of their micro-architectural design. The architecture targeted by this work (shown in Figure 1) is representative of the above mentioned platforms and composed by a many-core accelerator and an ARM-based processor.

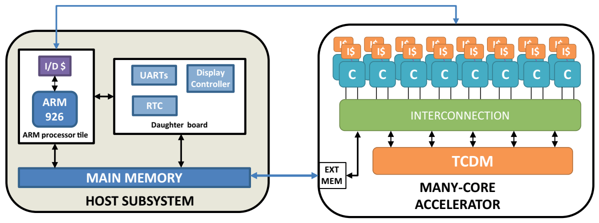

Figure 1: Target simulated architecture

The ARM processor is emulated by QEMU which models an ARM926 processor, featuring an ARMv5 ISA, and interfaced with a group of peripherals needed to run a full-fledged operating system (ARM Versatile Express baseboard). The many-core accelerator is a SystemC cycle-accurate MPSoC simulator. The ARM processor and the accelerator share the main memory, used as communication medium between the two. The accelerator target architecture features a configurable number of simple RISC cores, with private or shared I-cache architecture, all sharing a Tightly Coupled Data Memory (TCDM) accessible via a local interconnection. The state-of-the-art programming model for this kind of systems is very similar to the one proposed by OpenCL [22]: a master application is running on the host processor which, when encounters a data or task parallel section, offloads the computation to the accelerator. The master processor is in charge also of transferring input and output data.

## IV. Many-core Accelerator

The proposed target many-core accelerator template can be seen as a cluster of cores connected via a local and fast interconnect to the memory subsystem. The following subsections describe the building blocks of such cluster, shown in Figure 2.

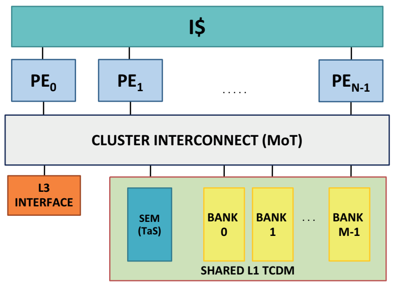

Figure 2: Many-core accelerator

*Processing Elements:* the accelerator consists of a configurable number of 32-bit RISC processor. In the specific platform instance that we consider in this paper we use ARMv6 processor models, specifically the ISS in [23]. To obtain timing accuracy we modified its internal behavior to model a Harvard architecture and we wrapped the ISS in a SystemC [24] module.

*Local interconnect:* the local interconnection has been modeled, from a behavioral point of view, as a parametric Mesh-of-Trees (MoT) interconnection network (*logarithmic interconnect*) to support high-performance communication between processors and memories resembling the hardware module described in [25], shown in Figure 3. The module is intended to connect processing elements to a multi-banked memory on both data and instruction side. Data routing is based on address decoding: a first-stage checks if the requested address falls within the local memory address range or has to be directed to the main memory. To increase module flexibility this stage is optional, enabling explicit L3 data access on the data side while, on the instruction side, can be bypassed letting the cache controller take care of L3 memory accesses for lines refill. The interconnect provides fine-grained address interleaving on the memory banks to reduce banking conflicts in case of multiple accesses to logically contiguous data structures. The crossing latency consists of one clock cycle. In case of multiple conflicting requests, for fair access to memory banks, a round-robin scheduler arbitrates access and a higher number of cycles is needed depending on the number of conflicting requests, with no latency in between. In case of no banking conflicts data routing is done in parallel for each core, thus enabling a sustainable full bandwidth for processors-memories communication. To reduce memory access time and increase shared memory throughput, read broadcast has been implemented and no extra cycles are needed when broadcast occurs.

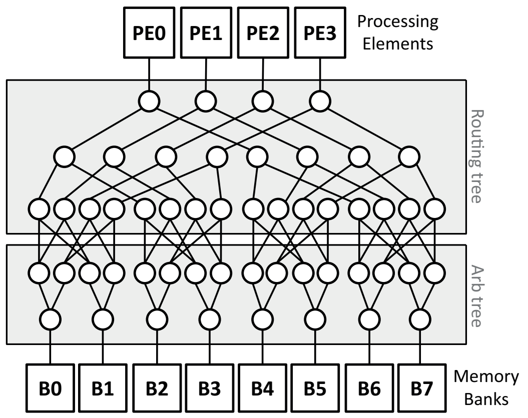

Figure 3: Mesh of trees 4x8 (banking factor of 2)

*TCDM:* On the data side, a L1 multi-ported, multi-banked, Tightly Coupled Data Memory (TCDM) is directly connected to the logarithmic interconnect. The number of memory ports is equal to the number of banks to have concurrent access to different memory locations. Once a read or write request is brought to the memory interface, the data is available on the negative edge of the same clock cycle, leading to two clock cycles latency for conflict-free TCDM access. As already mentioned above, if conflicts occur there is no extra latency between pending requests, once a given bank is active, it responds with no wait cycles.

*Synchronization:* To coordinate and synchronize cores execution the architecture exploits *HW semaphores* mapped in a small subset of the TCDM address range. They consist of a series of registers, accessible through the data logarithmic interconnect as a generic slave, associating a single register to a shared data structure in TCDM. By using a mechanism such as a hardware *test&set*, we are able to coordinate access: if reading returns '0', the resource is free and the semaphore automatically locks it, if it returns a different value, typically '1', access is not granted. This module enables both single and two-phases synchronization barriers, easily written at the software level.

*Instruction Cache Architecture:* the L1 Instruction Cache basic block has a core-side interface for instruction fetches and an external memory interface for refill. The inner structure consists of the actual memory and the cache controller logic managing the requests. The module is configurable in its total size, associativity, line size and replacement policy (FIFO, LRU, random). The basic block can be used to build different Instruction Cache architectures:

- *Private Instruction Cache*: every processing element has its private I-cache, each one with a separate cache line refill path to main memory leading to high contention on external L3 memory.
- *Shared Instruction Cache*: there is no difference between the private architecture in the data side except for the reduced contention L3 memory (line refill path is unique in this architecture). Shared cache inner structure is made of a configurable number of banks, a centralized logic to manage requests and a slightly modified version of the logarithmic interconnect described above: it connects processors to the shared memory banks operating line interleaving (1 line consists of 4 words). A round robin scheduling guarantees fair access to the banks. In case of two or more processors requesting the same instruction, they are served in broadcast not affecting hit latency. In case of concurrent instruction miss from two or more banks, a simple bus handles line refills in round robin towards the L3 bus.

## V. Host-Accelerator Interface

In this section we describe the QEMU-based host side of VirtualSoC (*VSoC-Host*), as well as the many-core accelerator side (*VSoC-Acc*).

*Parallel Execution:* In a real heterogeneous SoC host processor and accelerator can execute in an asynchronous parallel fashion, and exchange data using non-blocking communication primitives. Usually the host processor, while running an application, offloads asynchronously a parallel job to the accelerator and goes ahead with its execution (Figure 4). Only when needed the host processor synchronizes with the execution of the accelerator, to check the results of the computation.

In our virtual platform the host processor system and the accelerator can run in parallel, with VSoC-Host and VSoC-Acc running on different threads: when the thread of VSoC-Acc starts its execution triggers the SystemC simulation. It is important to highlight that the VSoC-Acc SystemC simulation starts immediately during VSoC-Host startup, and the accelerator starts executing the binary of a firmware (until the shutdown) in which all cores are waiting for a job to execute.

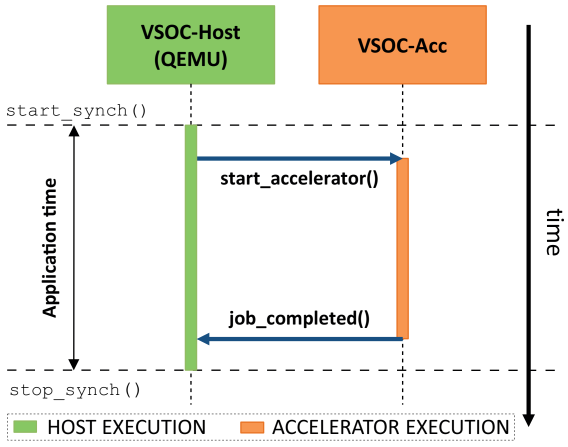

Figure 4: Execution model

*Time Synchronization Mechanism:* To manage the time synchronization between the two environments, it is necessary that both VSoC-Host and VSoC-Acc have a time measurement system. VSoC-Host does not natively provide this kind of mechanisms, so we instrumented it to implement a clock cycle count, based on instructions executed and memory accesses performed. On the contrary for VSoC-Acc there is no need for modifications because it is possible to exploit the SystemC time. The synchronization mechanism used in our platform is based on a threshold protocol acting on simulated time: at fixed synchronization points the simulated time of VSoC-Host and VSoC-Acc is compared. If the difference is bigger than the threshold, the entity with the biggest simulated time is stopped until the gap is filled. It is intuitive to note that the proposed mechanism slows down the simulation speed, due to synchronization points and depending on the difference of simulation speed between the two ecosystems. To avoid unnecessary slowdown, we provide an interface to activate and de-activate the time synchronization when it is not needed (e.g. functional simulation).

## VI. Simulation Software Support

In this section we provide a description of the software stack provided with the simulator to allow the programmer to fully exploit the accelerator from within the host Linux system, and to write parallel code to be accelerated.

*Linux Driver:* In order to build a full system simulation environment we mapped VSoC-Acc as a device in the device file system of the guest Linux environment running on top of VSoC-Host. A device node `/dev/vsoc` has been created, and as all Linux devices it is interfaced to the operating system using a Linux driver. The driver is in charge of mapping the shared memory region into the kernel I/O space. This region is not managed under virtual memory because the accelerator can deal only with physical addresses, as a consequence all buffers must be allocated contiguously (done by the Linux driver). The driver provides all basic functions to interact with the device.

*Host Side User-Space Library:* To simplify the job of the programmer we have designed a user level library, which provides a set of APIs that rely on the Linux driver functions.

Through this library the programmer is able to fully control the accelerator from the host Linux system. It is possible for example to offload a binary, or to check the status of the current executing job (e.g. checking if it has finished).

*Accelerator Side Software Support:* The basic manner we provide to write applications for the accelerator is to directly call from the program a set of low-level functions implemented as a user library, called appsupport. appsupport provides basic services for memory management, core ID resolution, synchronization. To further simplify programming and raise the level of abstraction we also support a fully-compliant OpenMP v3.0 programming model, with associated compiler and runtime library.

## VII. Evaluation

In this section two use cases of the simulation platform object of this work are presented. We will show how the proposed virtual platform can be exploited for both software verification or design space exploration.

### A. Experimental Setup

Table I summarizes the experimental setup of the virtual platform used for all benchmarks discussed.

Table I: Experimental Setup

| PARAMETER | VALUE |
|---|---|
| PLATFORM | |
| L3 latency | 200 ns |
| L3 size | 256 MB |
| ACCELERATOR | |
| PE | 16 |
| frequency | 250 MHz |
| L1 $I\$$ size | 16 KB |
| $t_{hit}$ | = 1 cycle |
| $t_{miss}$ | $\geq$ 50 cycles |
| TCDM banks | 16 |
| TCDM size | 256 KB |
| HOST | |
| ARM Core clock frequency | 1GHz |
| Guest OS | Debian for ARM (Linux 2.6.32) |

We chose as ARM core clock frequency of 1GHz, even if the ARM modeled by QEMU works at up to 500MHz, to resemble a state of the art ARM processor performance. The frequency would only affect results in terms of global values, all considerations done in this section remain valid even if the ARM core clock frequency is changed.

### B. VirtualSoC Use Cases

*Full System Simulation:* As first use case of the simulator we propose the profiling of an application involving both the ARM host and the many-core accelerator. In this example we want to measure the speedup achievable when accelerating a set of algorithms onto the many-core accelerator. The algorithms chosen are: *Matrix Multiplication*, *RGBtoHPG* color conversion, and *image rotation* algorithm. All the benchmarks follow a common scheme: the computation starts from the ARM host which in turn will offload a parallel task, one of the algorithms, to the accelerator. Then we compare simulated time obtained varying the number of cores present in the accelerator, with the time taken to run each benchmark on the ARM processor only (i.e. no acceleration).

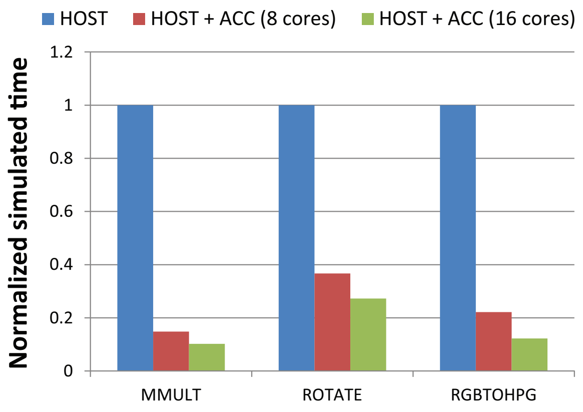

Figure 5: Speedup due to accelerator exploitation

Figure 5 shows the results of this experiment. Using the accelerator with 8 cores we can see a speedup of $\approx 3\times$ times for the matrix multiplication, $\approx 3\times$ for the rotate benchmark and $\approx 5\times$ for the RGBtoHPG benchmark. When running with 16 cores we can appreciate an almost double execution speedup for all the proposed benchmarks.

*Standlone Accelerator Simulation:* In this section we show an example of stand-alone accelerator analysis by using two real applications, namely a JPEG decoder and a Scale Invariant Feature Transform (SIFT), a widely adopted algorithm in the domain of image recognition. Our analysis will as first evaluate the effects of L3 latency over the execution time of each benchmark. In a second experiment we evaluate the instruction cache usage made by each application in terms of hit rate and average hit time. Fig. 6 shows the execution time when varying the L3 latency, and as expected the time increases when increasing the external memory access latency. The instruction cache utilization is shown in Fig. 7, depending on the application parallelization scheme the hit rate changes as well as the average hit time. The JPEG benchmark has been implemented in two different schemes: a data parallel implementation and a pipelined implementation. Results show that the data parallel version is more efficient in terms of cache hit rate and globally in terms of execution time. A deeper analysis has already been conducted on the aforementioned benchmarks, for further informations refer to [26].

## VIII. Conclusions

VirtualSoC leverages QEMU to model a ARMv6 host processor, capable of running a full-fledged Linux operating system. The many-core accelerator is modeled with higher accuracy using SystemC. We extended this combined simulation technology with a mechanism to allow for gathering timing information that is kept consistent over the two computational sub-blocks. A set of experiments over a number of representative benchmarks demonstrate the functionality, flexibility and efficiency of the proposed approach.

## Acknowledgment

This work was supported by the FP7 project VIRTICAL (288574).

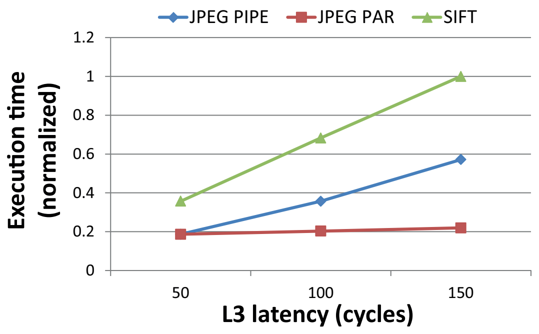

Figure 6: Benchmarks execution for varying L3 access latency (shared I-cache architecture)

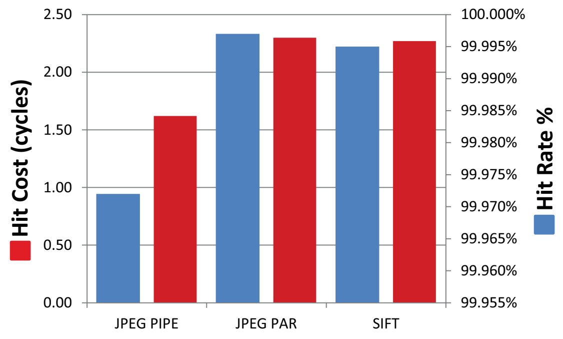

Figure 7: Benchmarks hit rate and average hit cost

## References

[1] N. Brookwood, "Amd fusion family of apus: enabling a superior, immersive pc experience," *Insight*, vol. 64, no. 1, pp. 1–8, 2010.

[2] NVidia Corp., "NVIDIA Tegra Multi-processor Architecture," 2010.

[3] Qualcomm Inc., "Snapdragon s4 processors: System on chip solutions for a new mobile age," 2011.

[4] K. Lawton, "Bochs: The open source ia-32 emulation project," *URL http://bochs. sourceforge. net*, 2003.

[5] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50 –58, feb 2002.

[6] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang, "Mambo: a full system simulator for the powerpc architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 8–12, Mar. 2004.

[7] R. Lantz, "Fast functional simulation with parallel embra," in *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*. Citeseer, 2008.

[8] M. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, april 2007, pp. 23–34.

[9] B. Barnes and J. Slice, "Simnow: A fast and functionally accurate amd x86-64 system simulator," in *Tutorial at the IEEE International Workload Characterization Symposium*, 2005.

[10] The Open Virtual Platforms, "OVPSim." [Online]. Available: http://www.ovpworld.org/

[11] Soclib Consortium and others, "Soclib: an open platform for virtual prototyping of multi-processors system on chip," Technical report, Tech. Rep., 2008.

[12] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[13] C. Guillon, "Program instrumentation with qemu," in *1st International QEMU Users' Forum*, vol. 1, March 2011, pp. 15–18.

[14] A. Miettinen, V. Hirvisalo, and J. Knuttila, "Using qemu in timing estimation for mobile software development," in *1st International QEMU Users' Forum*, vol. 1, March 2011, pp. 19–22.

[15] S. Raghav, A. Marongiu, C. Pinto, D. Atienza, M. Ruggiero, and L. Benini, "Full system simulation of many-core heterogeneous socs using gpu and qemu semihosting," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. New York, NY, USA: ACM, 2012, pp. 101–109.

[16] C. Pinto, S. Raghav, A. Marongiu, M. Ruggiero, D. Atienza, and L. Benini, "Gpgpu-accelerated parallel and fast simulation of thousand-core platforms," in *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGRID '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 53–62.

[17] M. Gligor and F. Petrot, "Combined use of dynamic binary translation and systemc for fast and accurate mpsoc simulation," in *1st International QEMU Users' Forum*, vol. 1, March 2011, pp. 19–22.

[18] D. Quaglia, F. Fummi, M. Macrina, and S. Saggin, "Timing aspects in qemu/systemc synchronization," in *1st International QEMU Users' Forum*, vol. 1, March 2011, pp. 11–14.

[19] M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina, "Mixed sw/systemc soc emulation framework," in *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, june 2007, pp. 2338 –2341.

[20] J.-W. Lin, C.-C. Wang, C.-Y. Chang, C.-H. Chen, K.-J. Lee, Y.-H. Chu, J.-C. Yeh, and Y.-C. Hsiao, "Full system simulation and verification framework," in *Information Assurance and Security, 2009. IAS '09. Fifth International Conference on*, vol. 1, aug. 2009, pp. 165 –168.

[21] T.-C. Yeh and M.-C. Chiang, "On the interfacing between qemu and systemc for virtual platform construction: Using dma as a case," *J. Syst. Archit.*, vol. 58, no. 3-4, pp. 99–111, Mar. 2012.

[22] Khronos OpenCL Working Group and others, "The opencl specification," *A. Munshi, Ed*, 2008.

[23] C. Helmstetter and V. Joloboff, "Simsoc: A systemc tlm integrated iss for full system simulation," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, 30 2008-dec. 3 2008, pp. 1759 –1762.

[24] "SystemC 2.3.0 Users Guide," 2012.

[25] A. Rahimi, I. Loi, M. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, march 2011, pp. 1–6.

[26] D. Bortolotti, F. Paterna, C. Pinto, A. Marongiu, M. Ruggiero, and L. Benini, "Exploring instruction caching strategies for tightly-coupled shared-memory clusters," in *System on Chip (SoC), 2011 International Symposium on*, 31 2011-nov. 2 2011, pp. 34 –41.