# A Framework for Dynamic Real-Time Reconfiguration

João Gabriel Reis, Antônio Augusto Fröhlich, and Lucas Wanner
Software/Hardware Integration Lab, Federal University of Santa Catarina
PO Box 476, 88040-900 - Florianópolis, SC, Brazil
{jgreis,guto,lucas}@lisha.ufsc.br

***Abstract*—In this work, we propose a framework capable of transparently switching between multiple hardware and software implementations of embedded system components to cope with and adapt to dynamic runtime characteristics such as power, throughput and quality of service. The reconfiguration process is decomposed into small steps such that it is preemptable, transparent, dynamic and compliant with real-time requirements. We present a Private Automatic Branch eXchange (PABX) system as a case study for the framework, and investigate the dynamic reconfiguration of three of its components: an ADPCM codec, a DTMF detector, and an AES core. Our results with this case study show how our framework is able to perform reconfiguration of hardware/software components in the order of a few milliseconds, without taking excessive system resources, and without disrupting the execution of application threads.**

***Keywords*—*Dynamic partial reconfiguration, Real-Time, Field-programmable gate arrays (FPGAs), System-level design, HW/SW co-design, High-level synthesis.***

## I. Introduction

Rigid partitions of components or modules in a hardware/-software co-design flow can lead to suboptimal choices in systems with dynamic or unpredictable runtime requirements. Partial reconfiguration allows systems to change portions of a hardware circuit (typically implemented on an FPGA) dynamically while other parts of the circuit are still active. Reconfiguration can help systems cope with dynamic non-functional requirements (such as performance and power), hardware defects (e.g. due to NBTI or PVT variations), or application requirements unforeseen at design time.

In this paper, we introduce a novel ROS whereby partial reconfiguration is performed automatically and transparently, and is compliant with real-time requirements. Components in our system are described through high-level models that can feature both hardware and software implementations. Functions provided by a component can be called identically regardless of how that component is instantiated at any point in time. Software adapters implemented as aspect programs abstract component-to-component communication, and helper functions for each component perform state migration when transitioning from software to hardware and vice-versa.

Each real-time process (or thread) in our system dynamically creates (and destroys) any components it may need (e.g. a multimedia codec, or an AES encryption module). For components with hybrid software/hardware implementations, our scheduler opportunistically and speculatively monitors system load and performance parameters and performs reconfiguration. The reconfiguration process for each component in divided into small tasklets such that its largest atomic step can typically be performed within available system slack as long as processor utilization is under 100%.

## II. Related Work

Several efforts have been made to develop reconfigurable computing environments that can help developers to leverage FPGA's resources. The Erlangen Slot Machine (ESM) [1] is a reconfigurable computing platform based on uniform resource allocation for each reconfigurable module. Andrews et al. propose HybridThreads (HTthreads) [2], a system that allows the execution of software threads on a CPU and hardware threads on and FPGA simultaneously.

BORPH [3], FUSE [4] and ReconOS [5], extend the Linux kernel providing native support for FPGAs, treating them as computational resources instead of coprocessors. The SPREAD programming model [6] focuses on high throughput point-to-point streaming applications and presents a common software/hardware thread interface and unlike the other solutions, in SPREAD a thread can be set as reconfigurable and thus switch domain during runtime. SPREAD allows switching threads domain during runtime but, as in ReconOS, the decision of switching is delegated to the application programmer, and it is not clear if thread migration can deal with real-time constraints.

R3TOS [7] differs from previous approaches in its capability of allocating hardware tasks in any FPGA region and not only reconfigurable slots defined in design time. CAP-OS (Configuration Access Port Operating System) exploits reconfiguration in heterogeneous MPSoCs implemented in FPGAs [8]. The capability to reconfigure hardware resources, as well as the number of processors in the system, is not supported.

Our proposal uses the Unified Design methodology [9] to create a single description of interchangeable software and hardware components. As in ELUS [10], the infrastructure is based on EPOS [11] component framework metaprogram extending it to support the reconfiguration requirements. Unlike other works that treat hardware/software reconfiguration as a monolithic operation, we split it into small steps executed while the operating system is idle. Hence, even with small available idle time, reconfiguration can be carried transparently and comply with real-time requirements.

## III. Design and Implementation

In order to have a consistent approach to reconfigurable computing, a framework design supporting it must be created. The framework must manage the communication between components in different domains, supply the user with a high-level design flow for system synthesis and to perform component reconfiguration transparently. The EPOS component framework [11] was extended to provide support for dynamic reconfiguration as well as to ensure the safe execution of concurrent applications.

### A. Communication Model

The operating system must efficiently manage communication so that components can seamlessly communicate. To abstract communication patterns between software and hardware components, we employ an approach based on Remote Method Invocation (RMI) concepts from distributed object platforms [12]. This approach, based on channels, proxies and agents, does not impose a single communication subsystem. It can be used with different networking technologies, such as a bus, a Network-on-Chip (NoC), or a Direct Memory Access (DMA) engine. Such variability is related to choices regarding the hardware/software architecture of the chosen implementation platform and should not affect the high-level components. Finally, the resulting communication bandwidth is limited mostly by the chosen underlying communication technology.

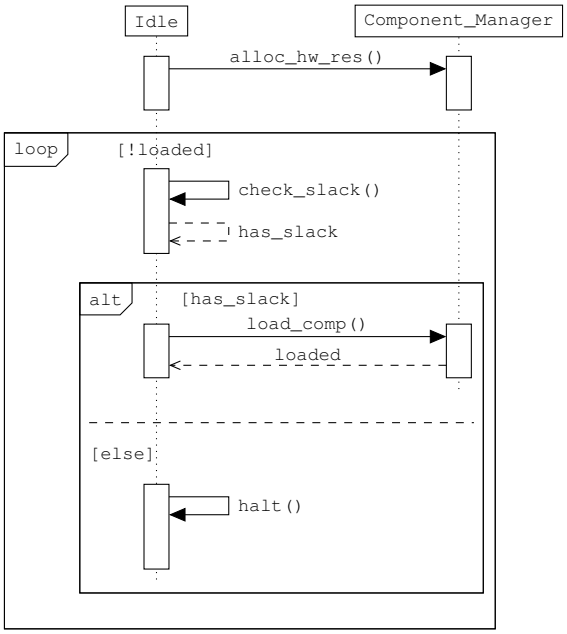
### B. Reconfiguration Process

Our approach assumes that concurrent applications are developed using threads. The reconfiguration process is performed by the operating system in a speculative manner based on the defined policy. EPOS hard real-time scheduler [13] ensures that the *idle* thread is only scheduled when there is no other thread to run. The speculative reconfiguration triggered by the idle thread is a key element in our approach, but ensuring that real-time threads will not have their deadlines compromised solely through it would require the reconfiguration procedure to be carried out without ever blocking the scheduler. Indeed, much of this procedure, which will be explained later on in details, can be performed in parallel with the execution of user threads, including reconfiguration policy enforcement and bitstream loading. A second feature of EPOS scheduler useful for our reconfiguration strategy is that it keeps the list of jobs–threads that are ready to be executed but did not yet reached their activation periods–in an ordered, relative queue. Therefore, calculating the amount of time available for atomic reconfiguration activities becomes a deterministic operation that simply consists in subtracting the time counter kept along with the queue's head from the current time.

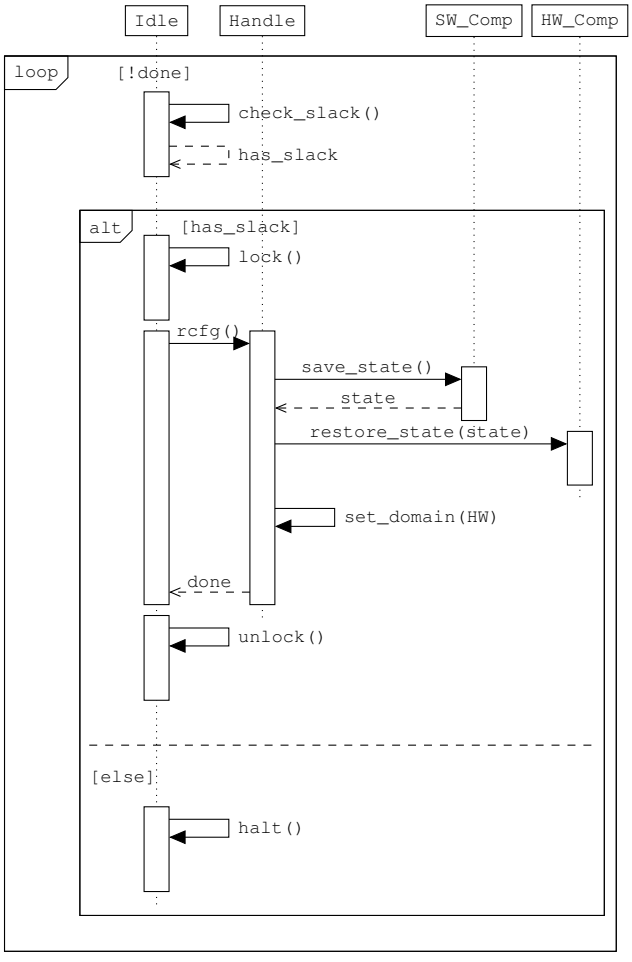
In order to explain the reconfiguration process in details, let us assume a reconfiguration policy that will try to push as much components to hardware as possible. The process would be comprised of two main steps, which are illustrated by the sequence diagrams in Figure 1. An operating system component called `Component_Manager` manages component reconfiguration and resource allocation.

In the first step of the process, presented in Figure 1a, the idle thread loads a given component into the reconfigurable hardware. Initially, the allocation of the hardware resources which will be used by the component is performed in `alloc_hw_res()`. Next, a bitstream containing the implementation of the given component to the available partition is loaded into the FPGA. Chunks of the bitstream are fetched and loaded to the bitstream each time `load_comp()` in invoked by the idle thread. The size of each chunk is adjusted to fit in the slack time available in the given scheduling period. `check_slack()` calculates if the available slack time is enough to transfer a minimal chunk of the component's bitstream. If the slack time is insufficient, the idle thread is halted. We therefore check for amount of time remaining until the next thread activation. The first step ends when the bitstream in completely loaded.

In order to finalize the process, `Handle` must also update its domain in order to dispatch the method invocations to the correct version of the component. This step, depicted in Figure 1b, must be atomically executed and therefore requires enough slack time. When `check_slack()` indicates that the slack time is enough, initially the idle thread applies a lock to the scheduler. One might argue that if the slack time is enough, the idle thread will not be preempted during the whole step thus the lock is pointless. Nevertheless, on multicore machines, the idle thread must acquire the lock to prevent cross-core interference. Next, the idle thread invokes the `rcfg()` method of the chosen component's `Handle`. It first fetches the software component's state through `save_state()` to further transfer it to the hardware component using `restore_state()`. As our components have a single high-level description for hardware and software implementations, their state in both domains can be captured by a group of internal variables. `save_state()` will send them upon request and the complementary method `restore_state()` will restore the internal state using the variables previously saved. At this point the




(a) Reconfiguration triggering and component loading.




(b) State handling and rebinding.

Figure 1: Reconfiguration process sequence diagram.

hardware component is ready to operate and its `Handle` change domain using `set_domain()`. From now on, the method invocations will all be dispatched to the component adapted to the hardware scenario.

The whole process is deterministic as we know the Worst Case-Execution Time (WCET) of each method. For most methods such as `check_slack()`, `alloc_hw_res()`, `set_domain()`, `lock()`, `unlock()` and `halt()` the process of obtaining its WCET is straightforward. Statistical methods based on static code analysis techniques and code profiling can be employed [14]. Nevertheless, for methods that rely on strict interaction with peripherals, the estimation might be trickier. For instance, `restore_state()` and `save_state()` execution time are component specific. It will vary according to the complexity of the component's internal state and bus access time but the WCET can also be estimated using statistical methods. As for `load_comp()`, it will be impacted by the underlying circuitry being used to reconfigure the fabric; Xilinx's commercial FPGAs rely on Internal Configuration Access Port (ICAP) or Processor Configuration Access Port (PCAP) while Altera's use Fast Passive Parallel (FPP). The WCET estimation for `load_comp()` will be based not only on the statistical methods mentioned previously but also on the reconfiguration bandwidth of the reconfiguration circuitry.

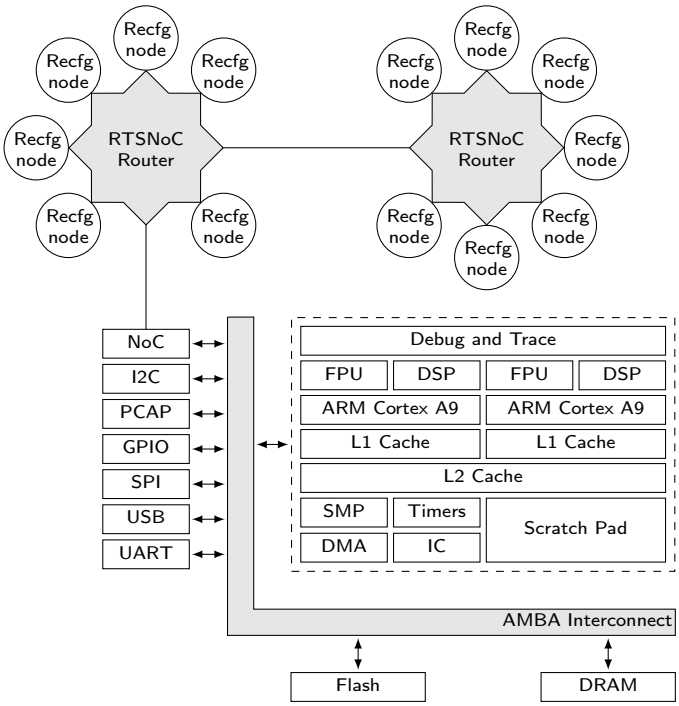



Figure 2: Block diagram of the system.

### C. Implementation

We assembled a platform called EPOSSoC to support the deployment of the reconfigurable components, Figure 2 shows its architecture. *Recfg nodes* represent the reconfigurable partitions that can contain a component in the hardware domain. We have chosen a Real-Time Star Network-on-Chip (RTSNoC) based interconnection scheme for channels, proxies and agents implementation. A bus-based interconnect was discarded as it is not the most suitable choice for more heterogeneous designs in which hardware components have active roles [15]. The software components run on the dual-core Cortex-A9 processor executing EPOS, also known as *CPU node*. It provides the necessary run-time support to implement the proxies and agents described previously. Hardware reconfiguration is held by the Processor Configuration Access Port (PCAP) interface on the AMBA Interconnect, also managed by the operating system.

## IV. Case Study

As a case study, we have implemented a phone data processing pipeline that includes typical operations of a Private Automatic Branch eXchange (PABX) system. The codecs interfacing the phone lines present trade-offs between compression ratio and complexity thus depicting a favorable scenario for dynamic partial reconfiguration. Each phone line holding a conversation is implemented as a thread in EPOS; they are dynamically created according to the system load. The threads have the necessary components to perform the phone data processing, and all the components can be deployed in both hardware and software domains.

The codecs interfacing the phone lines present a data compression ratio that scales with the codec complexity [16] and the resources necessary to implement it. Moreover, the smaller the codec's data compression ratio, higher the bandwidth occupied by the call. As the available bandwidth limits the number of additional calls that can be carried by the PABX, it must be managed without exhausting other system resources according to system load. By sensing the bandwidth used by the codecs and the usage of hardware resources, the PABX can better adapt itself to the system load without compromising the ability to answer future calls. Such scenario is favorable for real-time dynamic reconfiguration as it is not possible to know how many calls will be held at a given time during system synthesis. Suppose a call was established initially using codec iLBC (Internet Low Bitrate Codec). If there are no available hardware resources at a given time and a higher priority service is scheduled to execute in hardware, codec iLBC might be exchanged by codec G.711, which presents smaller data compression ratio. The exchange will free hardware resources, and the operating system must be able to exchange codecs without missing deadlines otherwise, the absent codec generates line noise, degrading voice quality. Table I illustrates the previously described reconfiguration policy used in the PABX, it focuses on balancing the bandwidth and hardware resources usage.

Table I: PABX reconfiguration policy.

| Free bandwidth | Free hardware resources | Next calls | Reconfigure |
|---|---|---|---|
| High | High | Will use iLBC | Nothing |
| High | Low | Will use G.711 | iLBC into G.711 |
| Low | Low | Will use software codecs | Nothing |
| Low | High | Will use iLBC | G.711 into iLBC |

## V. Results

For the experimental evaluation we implemented the infrastructure and the platform described in the previous section. The operating system and application were compiled with GCC 4.4.4 targeting the ARMv7 ISA using level 2 optimization enabled by GCC's `-O2` flag. For the hardware flow, Calypto's CatapultC UV 2011a was used to obtain RTL descriptions of the components. The hardware platform was prototyped in a Xilinx's XC7Z020 SoC using Xilinx's Vivado 13.4 for RTL hardware synthesis. Synthesis constraints were adjusted on CatapultC and Vivado to minimize circuit area considering an operating frequency of 100 MHz.

Our PABX case study makes use of three different reconfigurable components: ADPCM codec, DTMF detector and AES core. In this experiment, as we are not addressing multi-core issues only one of the two Cortex-A9 cores is

enabled. Moreover, both L1 and L2 caches are disabled during experiment execution and bitstreams are stored in the DRAM. For each component, we have measured the execution time of each operation shown in Figure 1. The number of processor cycles needed to execute each operation was collected using the processor's cycle counter register. Next, the number of cycles was divided by the processor operating clock frequency, 667 MHz in order to obtain the execution time. Table II shows the gathered results; the first row presents the component name followed by the number of bytes of each partial bitstream containing the component's hardware implementation.

Table II: Execution time of each function call in the reconfiguration process of different reconfigurable components.

| | ADPCM (269,840 bytes) | | DTMF (241,384 bytes) | | AES (430,196 bytes) | |
|---|---|---|---|---|---|---|
| Operation | Time ($\mu$s) | Share | Time ($\mu$s) | Share | Time ($\mu$s) | Share |
| `alloc_hw_res()` | 10.40 | 0.50 % | 10.40 | 0.35 % | 10.40 | 0.03 % |
| `load_comp()` | 2,075.06 | 98.45 % | 1,855.29 | 63.15 % | 3,306.18 | 99.47 % |
| `lock()` | 0.66 | 0.03 % | 0.66 | 0.02 % | 0.66 | 0.02 % |
| `save_state()` | 0.01 | 0.00 % | 1,007.82 | 34.30 % | 6.10 | 0.18 % |
| `restore_state()` | 0.01 | 0.00 % | 63.20 | 2.15 % | 0.15 | 0.00 % |
| `set_domain()` | 0.01 | 0.00 % | 0.01 | 0.00 % | 0.01 | 0.00 % |
| `unlock()` | 0.42 | 0.02 % | 0.42 | 0.01 % | 0.42 | 0.01 % |
| Total | 2,086.57 | 100.0 % | 2,937.80 | 100.0 % | 3,323.92 | 100.0 % |

First, we note that the major part of the reconfiguration process is spent on the `load_comp()` method. It is important to observe that the ADPCM codec does not hold an internal state, thus its `save_state()` and `restore_state()` methods return almost immediately. Regarding the AES, its internal state consists of the encryption key it is using. For components that hold more complex internal states such as the DTMF detector, the number of cycles spent getting and setting the component state might rise. The DTMF detector has a buffer that can store hundreds of tone samples, when the buffer is filled the DTMF detection algorithm can be issued. All the tone samples held internally must be saved before reconfiguration and later restored. For the three components, the operations executed during the reconfiguration process have constant execution times and are thus, deterministic. Therefore, they can be easily incorporated in the operating system's idle thread when using a scheduler following a hard real-time policy or soft real-time policy such as the presented PABX.

As mentioned previously, the reconfiguration step depicted in Figure 1b must be executed atomically and depends on having enough slack time available for executing all operations. Of the three studied components, the DTMF detector is the one that needs more time to execute the last reconfiguration step. For instance, adding `lock()`, `save_state()`, `restore_state()`, `set_domain()` and `unlock()` execution times, a slack time of 1.07 ms is necessary. For multimedia systems like the PABX, slack times within this dimension are quite reasonable considering that audio sampling rates are a few kilohertz and the processors executing the operating system run at hundreds of megahertz.

## VI. Conclusion an future work

We presented a transparent approach to reconfigurable computing, geared towards the application programmer. In it, system components are described using high-level models that abstract hardware development to the user. We described all the operations performed to migrate a component from one domain to another and also profiled its execution time in our system. We implemented a PABX system in an FPGA based platform called EPOSSoCin order to show the feasibility of our approach. To test the infrastructure, we investigated the reconfiguration process of three components: ADPCM codec, DTMF detector and AES cryptographic standard core. Also, the slack time needed in real-time systems to implement each step of component dynamic reconfiguration is reasonable and fitted in our PABX test case.

Our future work is divided into two branches. First, we will investigate policies to guide the reconfiguration process and to choose when and which components need to be reconfigured based on the system requirements and incorporate it into our operating systems infrastructure. Second, the move towards nanometer technologies in SoC fabrication increases silicon susceptibility to aging effects and environmental changes causing errors in the SoC usage life. We are investigating how our framework might help to cope with multiple design objectives such as dependability, efficiency, and real-time operation by providing a transparent approach to partial reconfiguration.

## References


[1] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich, "The Erlangen slot machine: increasing flexibility in FPGA-based reconfigurable platforms," in *Proc. Intl. Conf. on Field-Programmable Technology*, 2005, pp. 37–42.

[2] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving Programming Model Abstractions for Reconfigurable Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 34–44, January 2008.

[3] R. Brodersen, A. Tkachenko, and H. K.-H. So, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," in *Proc. Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006, pp. 259–264.

[4] A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, May 2011, pp. 170–177.

[5] E. Lubbers and M. Platzner, "ReconOS: Multithreaded Programming for Reconfigurable Computers," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 1, pp. 8:1–8:33, October 2009.

[6] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, "SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model," *IEEE T. on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2179–2192, 2013.

[7] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, I. Martinez, T. Arslan, and J. Perez, "R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient, and Dependable Computing on FPGAs," *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1542–1556, August 2013.

[8] D. Göhringer, M. Hübner, E. N. Zeutebouo, and J. Becker, "Operating System for Runtime Reconfigurable Multiprocessor Systems," *International Journal of Reconfigurable Computing*, 2011.

[9] T. Mück and A. Fröhlich, "Towards Unified Design of Hardware and Software Components Using C++," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 2880–2893, 2013.

[10] G. Gracioli and A. A. Fröhlich, "ELUS: A dynamic software reconfiguration infrastructure for embedded systems," in *Proc. Intl. Conference on Telecommunications (ICT)*, April 2010, pp. 981 –988.

[11] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, August 2001, no. 17.

[12] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn, "Programming with Live Distributed Objects," in *Proceedings of the 22nd European Conference on Object-Oriented Programming*, 2008, pp. 463–489.

[13] G. Gracioli, A. Fröhlich, R. Pellizzoni, and S. Fischmeister, "Implementation and evaluation of global and partitioned scheduling in a real-time OS," *Real-Time Systems*, vol. 49, no. 6, 2013.

[14] R. Wilhelm and et. al., "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[15] G. De Micheli, C. Seiculescu, S. Murali, L. Benini, F. Angiolini, and A. Pullini, "Networks on Chips: from research to products," in *Proc. Design Automation Conference (DAC)*, 2010, pp. 300–305.

[16] D. Guide, O. Hersent, and J. P. Petit, *IP Telephony*, 1st ed. USA: Prentice Hall, 2002.