Network/System Co-Simulation for Design Space Exploration of IoT Applications

Zhuoran Zhao*, Vasileios Tsoutsouras†, Dimitrios Soudris† and Andreas Gerstlauer*
The University of Texas at Austin, Austin, TX, USA*
National Technical University of Athens, Greece†
Email: zhuoran@utexas.edu, {billtsou,dsoudris}@microlab.ntua.gr, gerstl@ece.utexas.edu

***Abstract*—With the growing complexity and scale of future Internet of Things (IoT) applications, there is a need for effectively exploring associated design spaces. IoT applications make use of inherently distributed processing. In such networks-of-systems (NoS), computation and communication is tightly coupled. Traditional design of networks and systems in isolation ignores how choices in one influence the other, and approaches for joint network/system co-exploration are lacking. In this paper, we propose a novel prototyping platform to enable comprehensive NoS design space exploration. Fast and accurate host-compiled system models are combined with a standard network simulator to provide a unified network/system co-simulation framework. Furthermore, detailed models of network interfaces and protocol stacks are integrated into host-compiled system and OS models to allow accurately capturing of network and system interactions. We apply our NoS simulator to two case studies from smart camera and healthcare application domains, and we demonstrate benefits and opportunities for exploration and optimization in network/system co-design. Results indicate that, depending on application, network and system configurations, application throughput can vary by an average of 26%, while device core utilization can vary by as much as 130%. These results confirm strong network and system interactions that could not be observed or optimized without novel co-simulation tools.**

I. Introduction

In the Internet of Things (IoT), physical objects are embedded with electronic systems and network connectivity. This novel computing paradigm is inherently characterized by dynamic distribution of data collection, aggregation and processing tasks among networked devices [1], [2]. As a result, computation and communication workloads are tightly coupled, which brings new challenges and opportunities for the co-design of both the system devices themselves and the networks composed out of them. In such a network-of-system (NoS) environment, the overall performance is not only influenced by individual factors such as available application parallelism, system resource constraints and network communication capabilities alone, but also by the interactions between them. During execution, computation and communication can be dynamically parallelized or serialized at intra- or inter-device level. In order to accurately estimate application performance, the potential overlap among each data processing and communication tasks need to be carefully considered. This is in turn a function of their realistic execution durations, dependencies and concurrency available in the underlying computation platform. There exist non-obvious tradeoffs, where determining how to optimally partition and offload application tasks among an appropriately designed NoS architecture requires a comprehensive consideration of design parameters from applications, processing platforms all the way to network configurations.

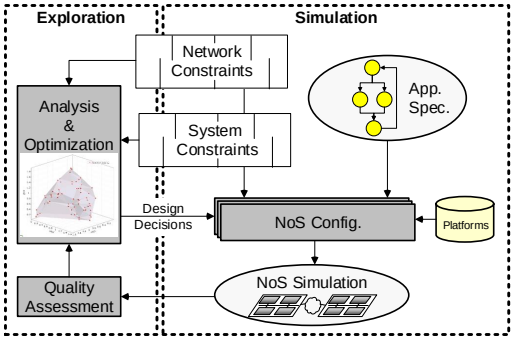


Fig. 1. Network/system co-design flow.

Exploring such design spaces requires novel approaches for systematic and joint network/system co-design that currently do not exist. An overview of such an envisioned NoS design process is shown in Fig. 1. Starting from an application specification and network and system constraints, the goal is to derive an optimal NoS architecture and application-to-architecture mapping. This is inherently an iterative process that requires fast and accurate methods for NoS validation and prototyping. Due to the complex and dynamic nature of NoS, a seamless co-simulation of networks and systems will need to be a key method, where computation and communication delays are accurately simulated and their interactions are properly emulated. A flexible NoS simulation platform can instantiate different NoS configurations and generate corresponding simulation models, where systems, networks and their interactions can be simultaneously simulated. Simulation results are then feed back for performance analysis, after which corresponding design decisions for specific optimization targets such as performance, utilization or cost can be explored.

In this paper, we propose a novel virtual NoS prototyping platform. Our NoS simulator integrates an advanced source-level, host-compiled system simulation with a state-of-art network simulator and accurate models of network interfaces and OS-level protocol stacks to enable fast, flexible and comprehensive network/system co-simulation. Using our simulator, interactions of system and network configurations and their influence on various performance metrics for different applications can be investigated. We demonstrate the capabilities of our co-simulation platform on two case studies of real-world IoT application scenarios from camera vision and healthcare domains in a wireless NoS setup. During exploration, key parameters such as the number of clients,

This work was supported in part by the National Science Foundation under Grant CSR-1421642, and in part by the E.C. funded program AEGLE under H2020 Grant Agreement No:644906

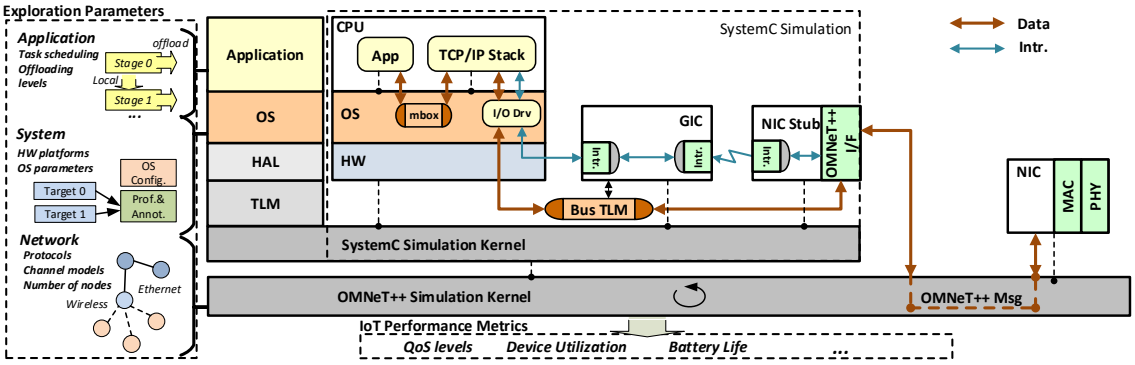


Fig. 2. Host-compiled NoS simulation platform.

application offloading levels, the number and configuration of processor cores, devices and radio/network interfaces are explored simultaneously. Results show non-obvious trade-offs coming from the interactions of parameters at various design levels, which can not be discovered from existing isolated network or systems simulations.

The rest of this paper is organized as follows: In Section II, related works on system and network simulation and IoT application trends are discussed. Section III then describes details of our NoS simulator. IV introduces IoT applications used in our case studies, results and observations of which are summarized in Section V. Finally, we conclude the paper with a summary and outlook on future work in Section VI.

II. Related Work

Contemporary IoT applications are shifting from simple traditional wireless sensor networks (WSN) to more sophisticated scenarios featuring more powerful embedded systems, widely distributed data and a higher level of mobility. For example, in smart camera networks, computer vision methods are deployed on a set of distributed mobile cameras equipped with high-performance computing and communication architectures for video stream capturing, processing, and communication [3], [4]. Another example are healthcare systems. With the help of wearable mobile devices, biosignals can be easily collected and forwarded to gateways and servers for real-time monitoring and analysis [5], [6]. Such applications require a more comprehensive design method to simultaneously explore parameters from both system and network sides.

Existing approaches for simulation-based exploration of WSNs and networked systems are limited in their support for modeling possible interactions between systems and networks and their effects on application metrics, such as performance. Traditional network simulators model system devices as simple traffic generators or analytical models without considering internal details of system architectures [7], [8]. On the system side, a large body of work has recently investigated better abstractions for fast and accurate virtual platform prototyping of system architectures [9], using advanced techniques for transaction-level modeling (TLM) together with so-called source-level and host-compiled hardware/software simulation on top of standard system-level design languages, such as SystemC. Such approaches have been extended to include consideration of network effects [10], [11], but existing WSN-oriented simulators typically combine simple state-machine based system models with an overly simplified model of network protocols and channels [12], [13], [14], [15]. This leaves IoT design space exploration mainly focused on application partitions and device configurations [16], where optimization opportunities of communication and computation interactions in complicated NoS environments are not investigated.

III. NoS Simulator

An overview of the proposed NoS simulator is shown in Fig. 2. We base our simulator on the open-source, SystemC-based host-compiled system simulator from [17], which we extend and integrate with the OMNeT++ network simulator [7] to provide a comprehensive, flexible and fast co-exploration platform. Our co-simulation platform is designed to provide an easily reconfigurable modeling setup that allows instantiating different NoS application and architecture configurations to evaluate both functional and non-functional metrics, such as quality-of-service (QoS), performance or utilization.

Inside the host-compiled device models, system-wide interactions between applications tasks, the underlying OS and various hardware components communicating through a hardware abstraction layer (HAL) and device bus TLMs are captured. One host-compiled system model is instantiated for each NoS device on top of a SystemC simulation kernel. The SystemC kernel is further integrated into an overall OMNeT++ simulation backplane, which includes channel and communication models that capture the interactions between different system devices in a given network topology.

For accurate co-simulation, TCP/IP models in OMNeT++ are replaced by detailed TCP/IP protocol stacks integrated into the host-compiled device models. Networking service processes are instantiated as kernel mode tasks, which communicate with user application tasks using an OS-provided mailbox. A network interface card (NIC) hardware model is added to the simulated device, which is internally an empty stub that interfaces with a detailed model of the NIC's media access (MAC) and physical (PHY) layer realization in the OMNeT++ backplane. On the system side, the NIC model communicates with the protocol stack through interrupts and bus transactions using accurate models of generic interrupt controllers (GICs) and OS-level NIC drivers.

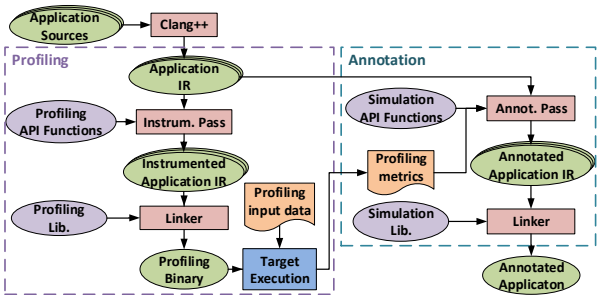


Fig. 3. Function-level profiling and back-annotation.

Fig. 2 highlights the data transmission path in dark orange while interruption propagation is in dark green. Upon arrival of a new network data packet, the NIC model in OMNeT++ will communicate with the NIC interface stub in SystemC system model through OMNeT++ messages, after which the NIC stub will buffer the data packet and generate an interrupt signal that is further distributed by the GIC model. The protocol stack process will then be triggered by the OS model's interrupt handling services and read the data packet through the TLM bus model from the NIC stub. For outgoing network data packets, the protocol stack can directly write into the NIC stub through the TLM bus, from where the packet will be forwarded to the OMNeT++ NIC through OMNeT++ message. In such a way, a SystemC device model can communicate with other device instances in an overall OMNeT++ network topology.

A. Network Simulation Model

We employ OMNeT++ together with the INET package [7] as our network simulation backplane. The OMNeT++/INET combination provides a flexible network simulation platform supporting rich choices of network protocols and topologies through an easy configuration process. In our setup, only the lower media access and physical layers are provided by OMNeT++ as part of the NIC models, which communicate with other OMNeT++ modules through wired or wireless channel models on the one side and forward network packets into/out of the simulated NIC stub on the other system side.

OMNeT++ natively provides a co-simulation with SystemC that integrates the two simulation kernels at the basic discrete-event level in a master-slave arrangement (with the OMNeT++ kernel serving as the master). During co-simulation, SystemC models are scheduled and synchronized globally by the OMNeT++ event scheduler. Through the NIC stubs, SystemC devices send and receive raw Ethernet packets to the OMNeT++ wrapper and invoke corresponding behaviors in the OMNeT++ network simulation. Conversely, the network simulator can access SystemC methods to notify occurrence of network events to the system model and NIC stub.

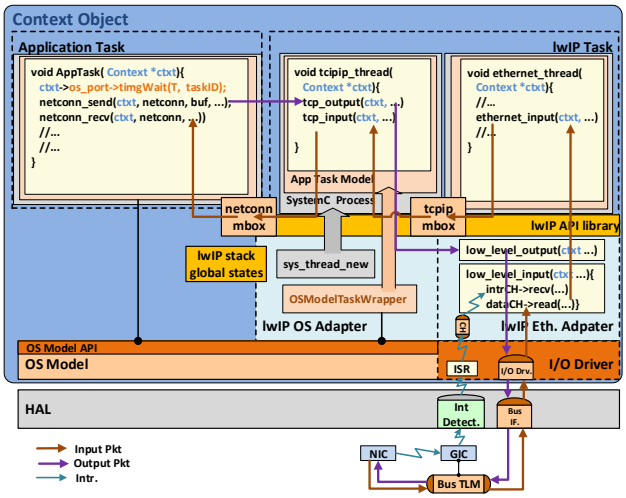
B. System Simulation Model

Existing host-compiled system simulators follow a layer-based organization to model the behavior and performance of complete multi-processor and multi-core systems-on-chip (SoCs) [17]. In such approaches, application source code is first instrumented with back-annotated performance metrics and then natively compiled and executed on the simulation host. This provides fast and accurate source-level simulation of raw application functionality and performance. Application task models are then mounted on top of a lightweight, abstract OS and processor model. OS and processor layers replicate a typical multicore OS and hardware architecture to emulate the execution of application tasks on a parallel hardware platform. Finally, host-compiled processor models are integrated with other CPU and hardware models using a standard TLM co-simulation backplane on top of an underlying SystemC kernel. In our NoS simulation setup, we extend such existing host-compiled simulators by integrating a lightweight TCP/IP stack in order to mount our IoT applications and allow integration within an overall network simulation.

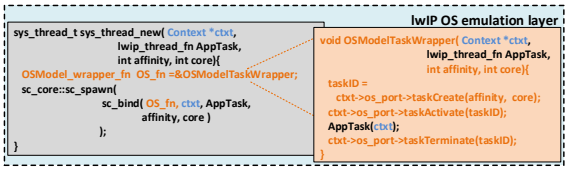
1) Application Task Model: Source code of IoT application tasks is first annotated with estimated execution time metrics and then converted into task models that run on top of the host-compiled OS simulator. The approach in [17] requires this to be done manually. We developed a customized LLVM pass to perform automatic back-annotation and conversion (Fig. 3). Our approach applies a function-level instrumentation by extract profiling metrics from running the application code on an existing hardware platform. Application source code is first converted into intermediate representation (IR) form. Profiling calls are inserted into the entry and exit points of each function to collect function-level performance metrics. The instrumented IR is then converted into an object file, linked with a profiling library, and the resulting binary is executed on the target platform under an input data set to collect cycle counts for all application functions. This profiling data is then used to back-annotate average per-function execution time estimates into the application code. Finally, application tasks are mapped onto a simulated system device by wrapping them into host-compiled task models that run on top of the OS API provided by the simulator.

2) Network Stack Model: In order to allow integration with network simulation while accurately accounting for associated system impact and overhead, we implemented a realistic model of the network stack and external network interfaces as part of the host-compiled system model. We ported a real network stack, *lwIP* [18], which is a widely-used, embedded and open-source TCP/IP stack, to our simulator and integrated it with host-compiled OS and hardware models, as shown in Figure 4. During integration, global variables from lwIP libraries are encapsulated in a context object, and all related lwIP function interfaces are modified to include a context object parameter representing corresponding system states, shown as the blue box in Figure 4a. Such context objects also include pointers to the corresponding OS and driver model. During simulation, the lwIP stack can then be instantiated as multiple copies along with different OS/device instances inside the NoS simulation environment. In our setup, the lwIP stack is also back-annotated with our profiling framework in order to estimate its core execution time.

lwIP provides a set of Netconn APIs for application process to create network connections and transfer data, as shown in



(a) Integration of lwIP on OS Model.



(b) lwIP OS emulation layer.

Fig. 4. Integration of lwIP.

the `AppTask` task model in Figure 4a. A multi-threaded configuration of the lwIP stack is ported onto the host-compiled simulator, where a protocol thread (`tcpip_thread`) is created for network packets processing, and a network interface thread (`ethernet_thread`) is created as an interrupt-driven task to read incoming network packets. The input and output data flows between different threads are highlighted in Figure 4a. The incoming raw network packets are transferred from the `ethernet_thread` to the `tcpip_thread` through a `tcpipmbox`, and from there processed and further delivered to application tasks through the `netconnmbox`. For outgoing networking data, the Netconn API will notify and deliver a message to the `tcpip_thread`, which will perform the protocol processing and ultimately send out packets using a low-level output function.

For porting to a new OS and hardware architecture, the lwIP stack relies on two thin OS and Ethernet adapter files that need to be implemented on top of existing OS and NIC driver APIs, respectively. We developed a custom driver for our NIC and customized the lwIP adapter files to port the stack onto our host-compiled OS, driver and NIC device models.

The lwIP OS adapter requires services for dynamic task creation and synchronization to be provided, which we implemented internally using host-compiled SystemC, OS and semaphore APIs. We rely on the `sc_core::sc_spawn` SystemC API and an OS task wrapper function, `OSModelTaskWrapper`, to provide the adapter's `sys_thread_new` function for dynamic task creation, as shown in the code snippet in Figure 4b. On every call to `sys_thread_new`, a new OS task wrapper is spawned dynamically as a SystemC simulation process by `sc_core::sc_spawn`. Inside the wrapper, OS model APIs are invoked to first register and mount the new SystemC process as application task on top of the OS model, and then execute the actual application task code (`AppTask`).

The lwIP Ethernet adapter includes all necessary low-level I/O functions to interface with the NIC hardware. With pointers provided by the context object, such I/O functions are able to access the related OS-level driver and interrupt services. We implemented a NIC driver that provides interrupt notification through an interrupt service routine (ISR) as well as APIs for exchanging packets with the NIC hardware over the TLM bus on top of host-compiled HAL and OS models. When the Ethernet adapter's `low_level_output` function is invoked, data is directly written out into the NIC model through the driver and bus, while invocation of the `low_level_input` function will cause the caller process to wait for an interrupt notification from the ISR before further reading the incoming data from the NIC model through the driver.

IV. IoT Application Case Studies

To demonstrate benefits and capabilities of our NoS simulation platform for network/system co-design and exploration, we apply our simulator to two case studies of vision graph discovery and ECG arrhythmia detection as representative IoT applications. These two applications have different computation/communication ratios, and in turn expose various patterns and sensitivities under system and network configuration changes. We map these applications onto different NoS architectures with one server and multiple client devices. The applications are augmented with tunable offloading parameters to formulate realistic usage scenarios for our case study.

A. Vision Graph Discovery

In a network of smart cameras, estimating their position and orientation relative to each other and to their environment is an essential operation and key challenge [3]. This type of relationship is defined as vision graph [19]. In a vision graph, each camera is represented by a node, and an edge appears between two nodes if the two cameras share a sufficiently large part of a view. Our formulation of different vision graph discovery scenarios is demonstrated in Fig. 5. Computation is divided into four stages, along which we define three offloading levels:

Image Capture (0) A stream of images is captured on each camera. In our case, we assume that enough images are already captured and buffered in the device memory.

Keypoint Extraction (1) For each image, a set of key locations (such as corners, blobs) that describe the image context for further scene analysis is extracted. In our setup, we use a Scale-Invariant Feature Transform (SIFT) algorithm to detect and describe such local image features [20].

Feature Matching (2) Keypoints between two images captured by different camera nodes are matched by identifying their nearest neighbours. The nearest neighbors are defined as the keypoints with minimum Euclidean distance from the corresponding keypoint descriptor vector.

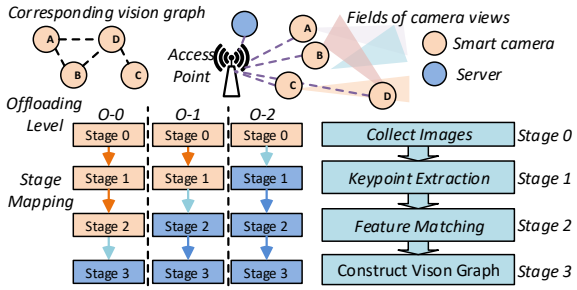


Fig. 5. Vision graph discovery.

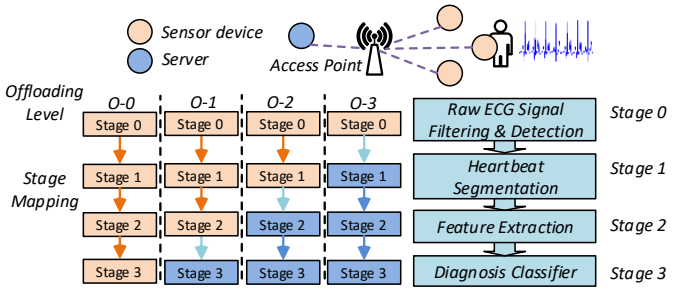


Fig. 6. ECG diagnosis.

Graph discovery (3) The numbers of matched points between every image pair are collected by the server, and a certain threshold is set for deciding the existence of view overlap between smart cameras. Based on this information, the server node can globally update the vision graph.

Possible offloading levels and corresponding task mappings for the vision graph application are shown in Fig. 5. Note that feature extraction requires image information from other nodes. Thus, when keeping stage 2 on the client nodes (offloading level 0), the server needs to first collect the output of stage 1 from all clients and then properly distribute the information. In our setup, we make a client node perform feature matching only with its adjacent client, and the server is responsible for matching all other possible image pairs.

We have implemented the vision graph application using OpenCV library. Video sequences from the Freiburg-Berkeley Motion Segmentation Dataset [21] are used as our input data.

B. ECG Monitoring and Diagnosis

In the healthcare domain, networked wearable devices providing continuous personal monitoring services are increasingly used to support early disease detection and improve the quality of healthcare. We formulate a case study in this domain using recent work on IoT-based ECG diagnosis optimization [16]. ECG signals are used to detect heart arrhythmia, i.e. irregularly fast or slow heart beats, which may lead to strokes or heart failure. As shown in Fig. 6, the detection flow is divided into four stages with four possible offloading levels:

Filtering and Heartbeat Detection (0) Raw ECG signal are captured by wearable devices, processed with a band-pass FIR filter to remove baseline wander and power line noise, and examined in a window to locate the R peak.

Heartbeat Segmentation (1) The detected R peak is used as the start of a new segment, and the ECG signals are segmented into single heartbeats for further analysis.

Feature Extraction (2) Feature extraction of the heart beat is performed through a Discrete Wavelet Transformation (DWT).

Diagnosis (3) A Support Vector Machine (SVM) classifier is used to determine whether the processed heart beat exhibits any signs of arrhythmia.

The ECG processing flow is implemented in C. Data from the MIT-BIH Arrhythmia Database [22] is used as input.

V. Experiments and Results

We demonstrate and validate our NoS simulator on the two IoT case studies presented in Section IV. For our experiments, we set up different NoS configurations in a wireless local area network (WLAN) client-server topology. The server device is configured as a dual-core system, with one core dedicated for networking-related service processes and another core for application tasks. Each edge node device is set up as a single-core system. Realistic execution time information is obtained using a Raspberry Pi 3 (RPi3) with a 1.2 GHz ARM Cortex-A53 and a Raspberry Pi 0 (RPi0) with a 1 GHz ARM11 as reference platforms for profiling and back-annotation. To extract timing information from the physical Raspberry Pi devices, we record cycle counters using PAPI. Network-related timing information is provided by the INET/OMNeT++ library.

The final NoS simulation setup is shown in Fig. 7. The ported lwIP in the host-compiled platform model replaces OMNeT++ TCP/IP models with an equivalent SystemC realization, and is back-annotated at the function level to accurately model its timing as well as the workload it incurs on the processor. In the OS model, lwIP service processes are instantiated as kernel mode tasks, which communicate with user application tasks using a mailbox protected by OS semaphores. The lwIP stack reads arrived network packets from the NIC data buffer driven by the OS model's interruption handling services. The lower media access and physical layers remain in OMNeT++ as part of a local NIC model. In the system node, the NIC interface stub (NIC Stub) is used to interact with the OMNeT++ NIC model through OMNeT++ events, and further configured to communicate with the lwIP driver through the bus model and interrupt services of the OS model. Finally, the whole SystemC system model is wrapped into an OMNeT++ node object to be synchronized globally by the OMNeT++ event scheduler.

A. NoS Design Space Exploration

We explore IoT applications under different implementation configurations. Scenarios with 2-10 clients are investigated. As detailed in Section IV, three possible offloading levels are defined for vision graph discovery (O-0, O-1 and O-2), and four for ECG diagnosis (O-0, O-1, O-2 and O-3).

We chose network configurations with three different MAC protocols and physical layer bandwidths from the 802.11 family, namely mode 'g' at 9Mbps or 54Mbps and mode 'b' at 11Mbps. The 802.11g and 802.11b protocols use different modulation schemes (OFDM and DSSS, respectively) with reduced protocol overhead and higher effective throughput in the 'g' mode. We further explore various client/server

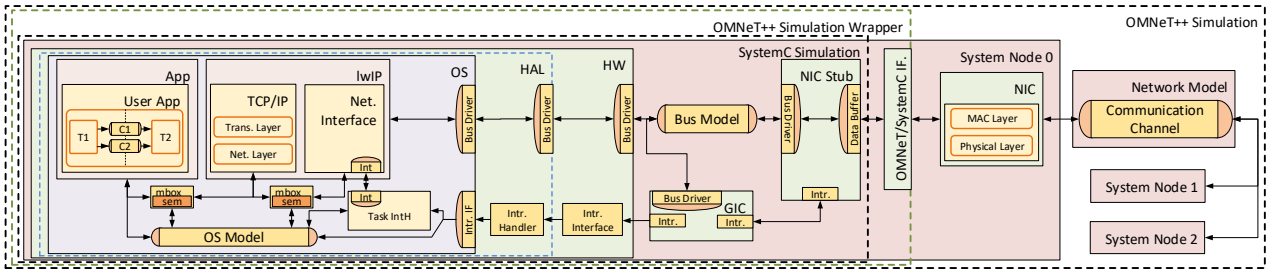
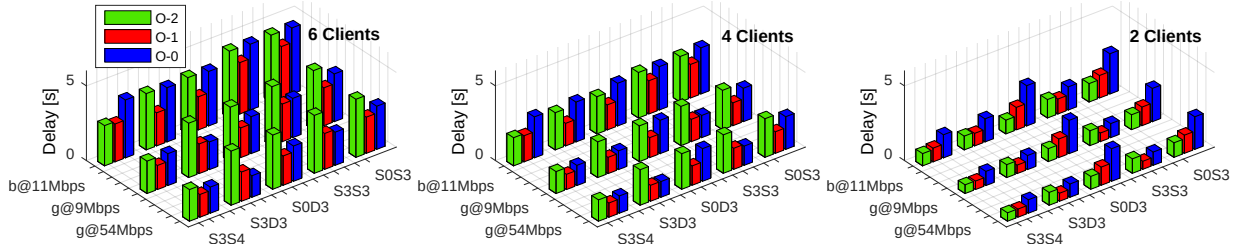
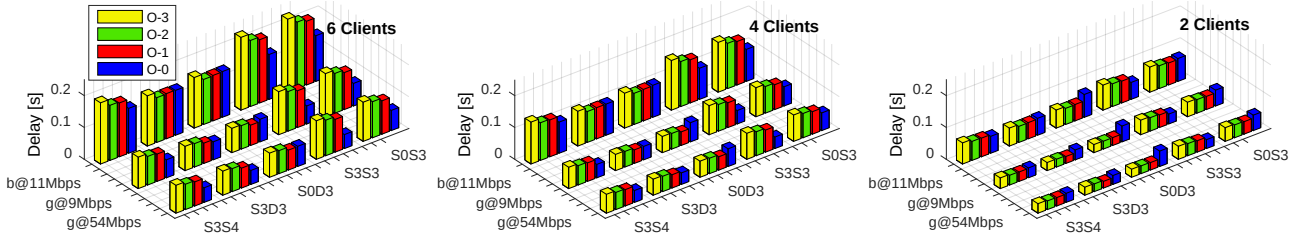


Fig. 7. Host-compiled NoS simulator setup for IoT application case studies.



(a) Vision graph discovery.



(b) ECG diagnosis.

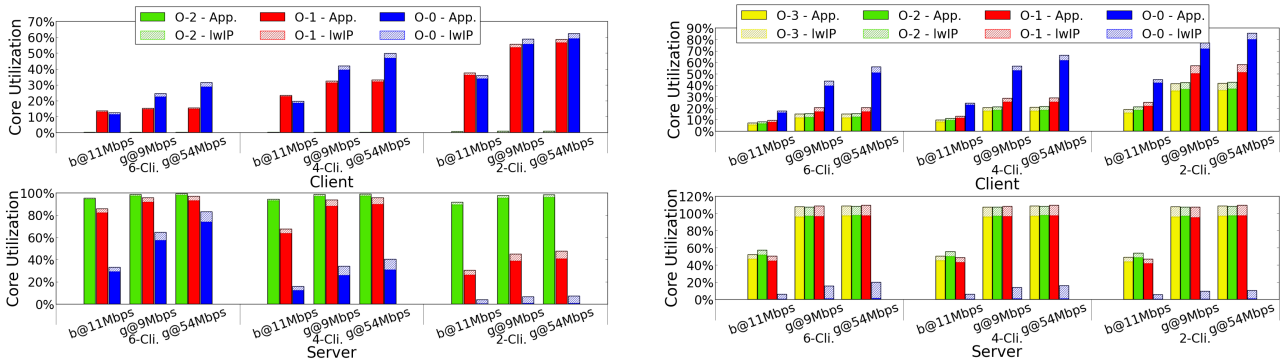
Fig. 8. Output-to-output delay of IoT applications.

configurations. Besides a single-core client/dual-core server setup (SD), a single-core client/single-core server setup (SS) is investigated. We use 0 and 3 indices to indicate the use of RPi0 or RPi3 cores, respectively. We introduce an additional virtual core type (denoted as 4) that assumes double the performance of a RPi3 core. In this naming convention, a single-core RPi0 client/dual-core RPi3 server setup will be represented as S0D3.

With this set of parameters, we examined different IoT performance metrics used for design space exploration. For the applications under study, all performance metrics are measured as an average among a fixed input stream window.

1) Throughput: We define throughput as the inverse of the output-to-output delay between successive data sets from all client devices, e.g., each newly detected heartbeat or captured image frame. Such a metric is a function of both communication/computation durations and dependencies. We investigated the interactions of parameters across different design levels and their effects on throughput. As results in Fig. 8 show, different application configurations show various sensitivities to system and network parameters. For both applications under study, scenarios with 2 to 6 client nodes are shown as representative cases. In both applications, computation and communication requirements and hence output delays increase with the number of clients. At the same time, delays generally decrease as a function of available network bandwidth and system processing power. However, relative ratios depend on the application, client count and offloading level.

The vision graph example (Fig. 8a) is generally computation-dominated, and network bandwidth has an overall smaller impact on performance than client/server configurations. Furthermore, server computation scales faster (quadratically) with client counts than communication and per-client demands (which grow linearly or are constant, respectively). As such, the application becomes server-bound for large numbers of clients and higher offloading levels. At 6 clients and offloading level O-2, neither network nor client configurations will significantly affect overall performance. With the dominance of computation, available server-side parallelism is also limited, which results in a relatively small benefit when going from a single-core (SxS3) to a double-core (SxD3) server. By contrast, a twice as powerful single-core server (S3S4) shows a more than 40% output delay reduction. With lower offloading levels, network and client configurations start to have a higher impact. Due to the need for centralized server coordination, both client computation and total communication demands increase with lower offloading in the vision graph example. At level O-0 with 6 clients, the server is mostly performing communication and coordination between clients, and throughput is noticeably affected by both client and network parameters. However, optimal balancing varies with the offloading level and depends on the specific interactions between client, server and network configurations. As the number of clients decreases, server and communication demands shrink, and the application becomes client-bound, especially at low offloading



(a) Vision graph discovery.

(b) ECG diagnosis.

Fig. 9. Core utilization of IoT applications.

levels. Thus, powerful client devices (S3Sx/S3Dx) can reduce the output delays by an average of more than 50% in the 2-client O-0 case. Effects otherwise depend on the application-specific ratios of server, client and communication demands as a function of offloading and client count.

In the ECG example, both computation and communication demands scale linearly with the number of clients, where communication overhead is more dominant and roughly constant, independent of the offloading level. As such, throughput generally varies with both network and system parameters at all offloading and client levels. In the case of O-3, O-2 and O-1 offloading, throughput is bounded by communication and server-side computation resources. As a result, network protocols can significantly affect overall output delay, and, in contrast to the vision graph case, a double-core server (SxD3) can reduce output delay by hiding server computation behind the communication overhead. In the O-0 case, throughput mainly depends on the client device configurations. However, as more clients are introduced, per-client computation remains constant, but communication overhead can dominate especially with slower communication protocols (b@11Mbps), where the effects of system device configurations become negligible.

2) System Core Utilization: Core utilizations for the S3D3 setup of our case studies are shown in Fig. 9, where Fig. 9a and Fig. 9b show the contribution of application and lwIP processing times to both client and total server load for vision graph and ECG applications, respectively.

In general, higher offloading levels will result in lower client and higher server utilization. For the ECG example, stage 3 is the most computation-intensive task with the largest relative impact when offloaded in levels higher than O-0. By contrast, stage 0 of the vision graph example adds negligible load when running as the only client task in O-2 offloading. With more clients, computation per client will remain the same while total throughput will decrease given larger communication and server overhead. As such, client utilization reduces as clients are added. By contrast, server utilization either remains constant or, in case of the vision graph example with quadratically scaling server demands, increases.

An interesting observation is that besides offloading levels and client counts, different network configurations can also

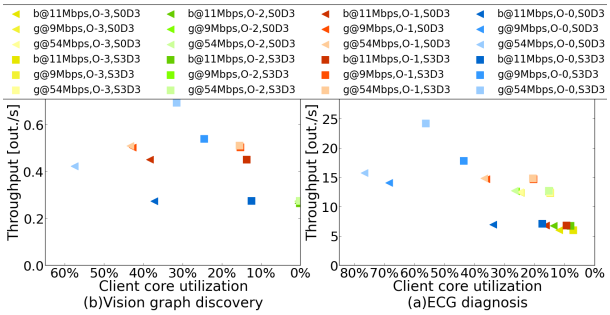


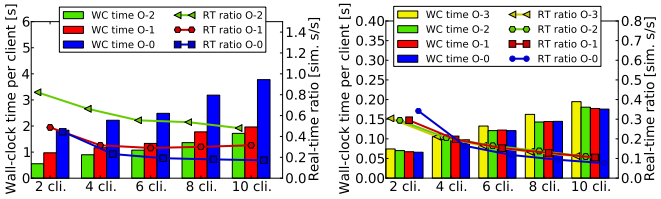
Fig. 10. Design trade-offs (6 client devices).

have a significant impact on core utilization due to their effect on communication overhead and thus throughput. On the client, the average core utilizations for vision graph discovery among different client counts and offloading levels range from 11.5% to 19.5% with various protocols. For ECG diagnosis they range between 13.7% to 31.6%.

Network stack overhead can contribute up to 20% core utilization depending on communication demands. In the vision graph case, communication decreases with increasing offloading level, with corresponding effects on lwIP overheads. In the ECG case, one server core is fully utilized by application computations at offloading levels above O-0, where the setup benefits from the second core to execute protocol functionality. Overall, this confirms the need to accurately model network stacks and their system impact.

3) Design Trade-offs: We further examine trade-offs between throughput and client device utilization. Fig. 10 shows the design points of a 6-client SD setup with different system and network parameters. In the graphs, colors indicate protocol choices, icon shapes represent different client core types, and offloading levels are distinguished by differences in shading.

As shown before, offloading influences client core utilization, with lower offloading levels (lighter shades) on the left. However, offloading also affects throughput, and other parameters affect both throughput and utilization in non-uniform ways, where design spaces can not be easily predicted without detailed exploration. Note that the Pareto front in both design spaces mainly consists of configurations with a S3D3 setup. This is because faster clients will improve throughput, but less than client computation time, thus also decreasing utilization. However, such a configuration will also have the highest cost,



(a) Vision graph discovery. (b) ECG diagnosis.

Fig. 11. Simulation wall-clock time and speed.

which will result in additional tradeoffs when considering other evaluation metrics and optimization objectives.

As these case studies show, IoT design space exploration involves multi-dimensional parameters at application, mapping and architecture levels with non-obvious interactions between each other. Fast NoS co-simulation for rapid prototyping at early design stages enables joint exploration across such application, network and system design spaces.

B. Simulation Speed and Accuracy

Fig. 11 shows the wall-clock time and speed of our simulations for different client numbers and offloading levels, averaged over all network and system configurations, which have a negligible effect on actual speed. Wall-clock time is shown as the average simulation runtime per input data set, normalized against the number of clients. The average simulation speeds for vision graph discovery and ECG diagnosis are 0.39 and 0.18 simulated seconds per wall-clock second, respectively. Vision graph discovery is about twice as fast since it is more computation-dominated, which requires relatively fewer OMNeT++ and SystemC communication and synchronization events to be simulated. For both examples, per-client wall-clock time is increased and speed is dropped with more clients, which is caused by extra synchronization and context switch overhead. In the vision graph example, server-side computation increases quadratically with more clients. This negatively affects per-client wall-clock time but not speed. On the contrary, due to the application becoming even more computation-dominated, speed can increase. Offloading generally does not affect total computation, but may influence the amount of communication to be simulated. In the vision graph example, higher offloading levels require less communication, which results in faster simulation speed and lower wall-clock time. In the ECG case, both computation and communication are constant or slightly increased with higher offloading, with corresponding effects on speed and wall-clock time.

Overall accuracy of our NoS co-simulation platform is determined by the accuracy of underlying system and network simulators. The system simulator in [17] is reported to be more than 95% accurate using an equivalent (and in their case manual) function-level back-annotation approach. Accuracy of the OMNeT++ network simulator depends on a wide range of factors, such as propagation and channel models, which are out of the scope of this paper to verify and validate. Nevertheless, for the purposes of exploration, relative accuracy is most important, where OMNeT++ has been widely used in academia and industry to perform corresponding studies.

VI. Summary and Conclusions

In this paper, we propose a comprehensive host-compiled NoS simulation infrastructure, in which system and network parameters at different design levels such as number of cores, network configurations and application offloading levels can be explored simultaneously. Two state-of-art IoT scenarios from typical application domains are investigated in a case study to show the capabilities of our proposed platform. Non-obvious interactions and dependencies demonstrated by our case studies motivate and confirm the need for systematic network/system co-design and development of associated tools. Unlocking associated optimization opportunities will require research into future co-design approaches. Towards these goals, we have released our NoS simulator and IoT case studies for download in open-source form at [23]. Future work includes further verifying simulator accuracy against a wider variety of real-world NoS design setups.

References

[1] J. Gubbi *et al.*, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Elsevier FGCS*, 29(7):1645-1660, 2013.
[2] F. Bonomi *et al.*, "Fog computing and its role in the internet of things," in *MCC*, 2012.
[3] D. Devarajan *et al.*, "Distributed metric calibration of ad hoc camera networks," *ACM TOSN*, 2(3):380-403, 2006.
[4] M. Quaritsch *et al.*, "Autonomous multicamera tracking on embedded smart cameras," *EURASIP JES*, 2007(1):1-10, 2007.
[5] C. Doukas *et al.*, "Bringing IoT and cloud computing towards pervasive healthcare," in *IMIS*, 2012.
[6] L. Catarinucci *et al.*, "An IoT-aware architecture for smart healthcare systems," *IEEE JIOT*, 2(6):515-526, 2015.
[7] A. Varga *et al.*, "An overview of the OMNeT++ simulation environment," in *Simutools*, 2008.
[8] NS-3 discrete-event network simulator. [Online]. Available: http://www.nsnam.org
[9] O. Bringmann *et al.*, "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems," in *DATE*, 2015.
[10] F. Fummi *et al.*, "A SystemC-based framework for modeling and simulation of networked embedded systems," in *FDL*, 2008.
[11] A. Banerjee *et al.*, "Transaction level modeling of best-effort channels for networked embedded devices," in *IESS*, 2009.
[12] J. Sommer *et al.*, "Sysifos: Systemc simulator for sensor and communication systems," in *Mobility*, 2009.
[13] L. S. Bai *et al.*, "Automated construction of fast and accurate system-level models for wireless sensor networks," in *DATE*, 2011.
[14] W. Du *et al.*, "IDEA1: a validated SystemC-based system-level deisgn and simulation environment for wireless sensor network," *EURASIP JWCN*, 2011.
[15] M. Damm *et al.*, "Using transaction level modeling techniques for wireless sensor network simulation," in *DATE*, 2010.
[16] F. Samie *et al.*, "Distributed QoS management for internet of things under resource constraints," in *CODES+ISSS*, 2016.
[17] P. Razaghi *et al.*, "Host-compiled multicore system simulation for early real-time performance evaluation," *ACM TECS*, 13(5s):1-26, 2014.
[18] lwIP - A Lightweight TCP/IP stack. [Online]. Available: http://savannah.nongnu.org/projects/lwip
[19] Z. Cheng *et al.*, "Determining vision graphs for distributed camera networks using feature digests," *EURASIP JASP*, 2007(1):220-220, 2007.
[20] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Springer IJCV*, 60(2):91-110, 2004.
[21] P. Ochs *et al.*, "Segmentation of moving objects by long term video analysis," *IEEE TPAMI*, 36(6):1187-1200, 2014.
[22] G. B. Moody *et al.*, "The impact of the MIT-BIH arrhythmia database," *IEEE EMB Magazine*, 20(3):45-50, 2001.
[23] NoSSim. [Online]. Available: https://github.com/SLAM-Lab/NoSSim