Network-level Design Space Exploration of Resource-Constrained Networks-of-Systems

ZHUORAN ZHAO, University of Texas at Austin
KAMYAR MIRZAZAD BARIJOUCH, University of Texas at Austin
ANDREAS GERSTLAUER, University of Texas at Austin

Driven by recent advances in networking and computing technologies, distributed application scenarios are increasingly deployed on resource-constrained processing platforms. This includes networked embedded and cyber-physical systems as well as edge computing in mobile applications and the Internet of Things (IoT). In such resource-constrained Networks-of-Systems (NoS), computation and communication workloads need to be carefully co-optimized yet are tightly coupled. How to optimally partition and schedule application tasks among an appropriately designed NoS architecture requires a simultaneous consideration of design parameters from applications and processing platforms all the way to network configurations. Traditionally, however, systems and networks are designed in isolation and combined in an ad-hoc manner, which ignores joint effects and optimization opportunities. To systematically explore and optimize NoS design spaces, a higher level of design abstraction on top of traditional system and network design is required.

In this paper, we propose a novel network-level design methodology for resource-constrained NoS optimization and design space exploration. A key component in such a design flow is fast yet accurate network/system co-simulation to rapidly evaluate NoS parameters with high fidelity. We first introduce a novel NoS simulator (NoSSim) that integrates source-level simulation models of applications with a host-compiled system simulation platform and a reconfigurable network simulation backplane to accurately capture system and network interactions. The co-simulation platform is further combined with model generation tools and a multi-objective genetic search algorithm to provide a comprehensive and fully automated NoS design space exploration framework. Finally, we apply our network-level design flow on several state-of-art IoT/mobile design case studies. Results show that NoSSim can achieve more than 86% simulation accuracy on average as compared to a real world edge device cluster, where sensitivities to various design parameters are faithfully captured with high fidelity. When applying our network-level design space exploration methodology, design decisions are automatically optimized, where non-obvious NoS configurations are discovered outperforming manually-designed solutions by more than 45%.

CCS Concepts: • Computing methodologies → Modeling and simulation; • Computer systems organization → Embedded systems; • Networks → Network simulations;

Additional Key Words and Phrases: Source-level simulation, networks-of-systems, design space exploration

ACM Reference Format:

Authors’ addresses: Zhuoran Zhao, University of Texas at Austin; Kamyar Mirzazad Barijough, University of Texas at Austin; Andreas Gerstlauer, University of Texas at Austin.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.
1539-9087/2020/1-ART1 $15.00
https://doi.org/10.1145/3387918

1 INTRODUCTION

A networked and distributed computing paradigm is increasingly deployed on resource-constrained processing platforms. Traditionally isolated embedded and cyber-physical systems are extended with networking capabilities to realize more complicated and collaborative functionalities. At the same time, in the Internet-of-Things (IoT) and mobile applications, edge devices are increasingly used for both data collection and processing, where application tasks are moved closer to the data sources instead of offloading all computation to a centralized server or the cloud. In all these cases, applications are characterized by dynamic, distributed data collection and task processing, where computation and communication workloads are tightly coupled. Moreover, given tight resource constraints, underlying network and computing platforms need to be carefully co-optimized and co-designed while considering design parameters across the full stack in an application-specific manner. This creates new challenges and opportunities for the design of such Networks-of-Systems (NoS). Mapping, scheduling and offloading of computations across an appropriately designed NoS architecture comes with non-obvious trade-offs that will greatly influence overall application performance and energy consumption, where a comprehensive consideration of design parameters from applications and processing platforms all the way to network configurations is required.

Traditionally, networks and systems are designed and optimized in isolation. Corresponding design methods and modeling tools usually focus on one side while oversimplifying the other. Traditional network design methods \cite{35} usually emphasize overall network aspects such as traffic throughput and transmission latency, where system nodes are often overly simplified as simple, regular or statistical traffic generators or sinks. By contrast, existing system-level design approaches \cite{19} have proposed systematic and potentially fully automated approaches to convert abstract specifications into low-level implementations for individual, isolated system devices, but they consider network aspects and network/system interactions as separate, external constraints that are not explicitly captured or only accounted for in an ad-hoc manner.

In this paper, we propose a novel network-level design flow for NoS design and synthesis under joint consideration of both network and system sides. Within such a flow, a network/system co-simulation platform integrating detailed system and network models and accurately capturing their dynamic interactions needs to be a key method for evaluation of design options. In prior work, we have developed a reconfigurable NoS simulator (NoSSim) for fast and accurate NoS performance evaluation \cite{48}. NoSSim combines source-level, host-compiled system simulation techniques \cite{33} with a comprehensive network simulation backplane. In this work, we extend NoSSim to provide a more comprehensive programming interface that eases porting, mounting and development of applications on top of the simulator while also including more communication protocols and additional estimation metrics.

Even with fast simulation methods, large and complicated NoS design spaces with non-obvious parameter tradeoffs across multi-dimensional design objectives make manually enumerating and exploring all possible design options infeasible and suboptimal. We further extend our network/system co-simulation platform into a comprehensive and automated design space exploration framework, where NoS design parameters are automatically instantiated into NoSSim simulation models and iteratively optimized with a multi-objective evolutionary algorithm. We employ several state-of-art case studies to evaluate NoSSim simulation speed and accuracy and to demonstrate the efficiency and effectiveness of the fully automated network-level design space exploration flow.

Comparing with previous works, the specific contributions of this paper are as follows:

(1) We extend NoSSim and improve its usability by providing a full set of standard socket programming interfaces and a novel context-aware simulation method to ease deployment

of complex multi-threaded libraries while adding more IoT-specific communication protocols and energy models (Sections 4.1 and 4.2).

(2) We introduce a comprehensive design space exploration flow to systematically and automatically explore and optimize NoS design decisions. Related NoSSim model generators are developed and seamlessly combined with a multi-objective genetic algorithm backplane (Section 5).

(3) We demonstrate our flow for fully automated NoS synthesis on previously manually explored case studies from the smart camera and health care domains as well as a distributed deep learning inference framework on top of IoT edge clusters [47]. In the process, the accuracy and fidelity of NoSSim is evaluated, and non-obvious NoS parameter interactions are discovered and further optimized with our flow (Section 7).

The rest of this paper is organized as follows: An overview of our proposed network-level design flow is first introduced in Section 2. In Section 3, related works on system and network simulation and NoS design space exploration are discussed. Section 4 then describes the details of NoSSim, based on which Section 5 introduces our proposed automated NoS design space exploration framework. We further introduce three IoT applications in Section 6, which are employed to evaluate NoSSim and demonstrate our design space exploration flow in Section 7. Finally, we conclude the paper with a summary and outlook on future work in Section 8.

2 NETWORK-LEVEL DESIGN FLOW

Figure 1 shows an overview of our proposed network-level design flow. Starting from an application specification and network/system constraints, different NoS design decisions are iteratively explored to derive an optimal NoS architecture and application-to-architecture mapping. Within such a flow, parameters from different design levels can be simultaneously explored and optimized considering multiple design objectives.

Due to the complex and dynamic nature of NoS, fast and accurate co-simulation of networks and systems needs to be a key component allowing different application mappings and system/network configurations to be rapidly evaluated. A flexible NoS design space exploration framework then instantiates different NoS, network and system architecture configurations, deploys generic middleware systems and generates corresponding simulation models to automatically find Pareto-optimal NoS implementations. Related model generation tools are included to evaluate different NoS configurations in the network/system co-simulation platform, where platform, network and middleware
Component models can be deployed to accurately emulate dynamic runtime effects, such as dynamic NoS resource management, task migration and offloading. With the help of a multi-objective optimization algorithm, design parameters such as task mappings, communication protocols and system device architectures can then be iteratively explored and optimized based on corresponding simulation results, and Pareto-optimal NoS implementations for a wide range of design metrics can be automatically generated.

3 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 sources and a higher level of mobility. For example, in smart camera networks, computer vision methods are deployed for distributed video stream capturing, processing, and communication [11, 32]. In healthcare systems, biosignals from wearable devices are collected and forwarded to gateways and servers for real-time monitoring and analysis [7, 12]. Recently, with increasing application of deep learning methods in such scenarios, different approaches have been proposed to adopt distributed Deep/Convolutional Neural Network (DNN/CNN) frameworks on mobile, edge or IoT devices [20, 23, 40, 47]. All these applications require a 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. Traditional network simulators model system devices as simple traffic generators or analytical models without considering internal details of system architectures [28, 41]. On the system side, a large body of work has recently investigated better abstractions for fast and accurate virtual platform prototyping [6, 18, 26], using advanced techniques for transaction-level modeling (TLM) together with source-level and host-compiled hardware/software simulation on top of standard system-level design languages. Some initial attempts have been proposed to extend TLM techniques to include consideration of network effects [3, 17], but existing approaches are limited in their capabilities for modeling of complex networks and emulation of full-system behavior including operating system (OS) and network stack effects. Existing WSN-oriented simulators typically combine simple state-machine based system models with an overly simplified model of network protocols and channels [1, 9, 13, 38]. This leaves design space exploration mainly focused on simple application partitioning and device configuration [36]. In our own prior work, we introduced a novel NoS simulator based on an existing host-compiled simulation platform [33] integrated with a widely used, state-of-the-art network simulator [41] to establish a complete network/system co-simulation and NoS evaluation method. In this paper, we extend this simulator into a complete and comprehensive NoS design space exploration framework.

Several design space exploration frameworks for WSNs have been proposed [2, 5, 24, 31], where different WSN setups for performance and energy optimization are explored using simple state-based system and traffic models. In [37], embedded system simulation is extended with network interface models to consider network effects, where different application design solutions are explored for minimal SoC energy consumption. A TLM-based refinement and exploration approach is further proposed in [4], where a new design dimension is added to the traditional TLM refinement process to represent network configuration alternatives. However, none of the above-mentioned works consider complex NoS scenarios or employ any automated exploration methods.

Population-based meta-heuristic methods such as evolutionary algorithms (EA) are widely used and the de-facto standard for exploring large and complex multi-objective design spaces [30]. Specifically, multi-objective genetic algorithms (MOGAs) [10, 49] are widely adopted in system-level designs [14, 21] for multi-objective design space exploration. In [39], the communication
Table 1. Comparison with state-of-the-art design approaches.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>System Modeling</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Network Modeling</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Full-system Capability</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Automated NoS DSE</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
</tbody>
</table>

topology of networked embedded systems is optimized using a GA for better application reliability. In the domain of WSN, MOGAs are employed to optimize multiple WSN design metrics such as latency, energy and reliability using abstract analytical models [15, 27]. In our work, we also base our automated exploration framework on a MOGA. Different from prior work, we encode parameters all the way from application to system and network levels into genomes, and we develop model generators to automatically instantiate complete NoS simulators for fitness assignment.

Table 1 summarizes the comparison of our proposed NoS design space exploration framework (NoSDSE) with other representative state-of-the-art network/system design approaches. To the best of our knowledge, NoSDSE is the first approach to enable fully automated NoS design space exploration considering detailed dynamic behavior of both system and network architectures including the capability for modeling of full-system OS and runtime system effects.

4 NOS SIMULATOR

An overview of the proposed NoS simulator is shown in Figure 2. We base our simulator on the open-source, SystemC-based host-compiled system simulator from [33], which we extend and integrate with the OMNeT++ network simulator from [41] 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 energy consumption.

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.

Figure 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.

4.1 Application Model

In order to deploy applications on the NoS simulator, source code of application tasks need to be first converted into task models that run on top of the host-compiled simulator. Figure 3 shows an example for mounting an application on top of our proposed NoS simulator. Detailed application model examples can be found in [45]. We provide a complete SystemC device model template encapsulated into an OMNeT++ simulation wrapper that users can instantiate and customize to define the nodes of their overall NoS architecture. The SystemC device template provided with our simulator internally pre-instantiates CPU, NIC, GIC and bus models, but can be customized, e.g. to include additional system architecture components such as hardware accelerators. Within the CPU, the template provides all the necessary HAL and OS layers with an OS model that launches a single initialization task (app_init()) from which additional application-specific user tasks can be filled in and invoked. Specifically, the initialization task app_init() in the provided device template will invoke an empty top-level function (app_start()) to be filled by the user with application functionality.

4.1.1 Task Model. To convert application source code into task models running on top of the simulator, users need to integrate tasks into the CPU model and port all the OS- and network-related operations onto corresponding API calls implemented by the simulator. To facilitate easy application mounting, we provide a full set of OS and network socket APIs following the specifications in lwIP’s system abstraction layer [22], which is widely adopted on various embedded platforms. Typical OS and network socket APIs provided by our simulation are shown on the right of Figure 3. As part of OS APIs, we include methods (sys_time_wait()) for source-level execution time annotation (see Section 4.1.2 below). Additionally, we provide a set of context APIs to enable multi-instance simulation. To allow for multiple, device-specific instantiations of the same application and library code, any global variables need to be encapsulated and wrapped into an application context class that can then be instantiated separately in different simulated devices. Further details on such context-aware simulation will be introduced later in Section 4.2.1.

A typical case of using application APIs is shown in Figure 3. The top level function (app_start()) first constructs and registers the application context (app_ctxt). All the application tasks are then created and started from there using sys_thread_new() APIs to create tasks (task) running on top of the OS model. Within the example task shown in the figure, a globally allocated mailbox
instance encapsulated by the application context (ctxt->mbox) is retrieved. In order to use the OS-provided mailbox for inter-thread communication and synchronization, corresponding APIs such as sys_mbox_new(), sys_mbox_fetch() and sys_mbox_post() are employed with application-level parameters. Furthermore, for network programming, socket APIs can be directly used in the application source code to create sockets, establish network connections and transmit data (lwip_socket(), lwip_accept() and lwip_send()).

### 4.1.2 Application Timing Models

For performance estimation, application task models need to be further annotated with estimated execution time metrics at certain granularity to record corresponding execution durations. The approach in [33] requires this to be done manually. Existing source-level simulation approaches typically perform back-annotation at the detailed basic block level [46]. In this paper, we apply a a customized LLVM pass to perform lightweight, automatic function-level back-annotation [48] for simplicity. In the process, we also create a variant of tasks models in which all functionality is removed and tasks are represented purely as empty functions without code but annotated timing only.

### 4.2 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) [33]. In such approaches, application source code is first back-annotated with 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 multi-core 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 context-aware OS and processor models for easy porting and mounting of multi-threaded applications and a lightweight TCP/IP stack in order to mount networked applications and allow integration within an overall network simulation.

#### 4.2.1 Context-Aware OS and Processor Models

In the OS model proposed in [33], each application task needs to be manually refined with the OS model APIs and encapsulated within an OS model task object. All task objects are then instantiated explicitly at design time within corresponding OS model contexts (one per simulated device or processor). Such a modeling mechanism assumes that the set of running threads is known a priori. Additionally, in the process, all function interfaces from

---

To address such problems and provide a generic and flexible simulation interface, we introduce a novel context-aware host-compiled simulation technique, as shown in Figure 4. A simulation context includes both an OS model pointer and the application context encapsulating application-specific global states as described in Section 4.1.1. Application threads can be dynamically created and associated with a context using generic multi-threaded programming interfaces. During execution, the simulation context can be dynamically retrieved using a global lookup table. In this way, an application and external library can be easily ported onto the OS model by modifying only OS-related functions calls for thread creation, message passing, etc. while the function interfaces and software architecture of the original libraries can be reused without any modifications.

In host-compiled system simulation, the instantiation of each individual OS model task requires a new simulation process from the underlying discrete-event simulation kernel as a container to wrap the task in. Instead of syntactically encapsulating the tasks into simulation processes, we introduce a global context table to enable execution-time context retrieval. The global context table, which is indexed by simulation kernel process handles, stores the necessary OS model and application contexts for correct functional emulation and accurate performance estimation.

For example, as shown in Figure 4, SystemC Process 6 virtually encapsulates OS model task 3, which further executes the code in application function app_task_1(). The app_task_1() function first queries the simulation kernel to get its simulation process handle, which is then used to fetch the corresponding OS model pointer, OS task ID and application context from the global context table. The OS model pointer can then be used to invoke OS model APIs, e.g. to manage timing information or perform thread-level synchronization. Such OS model-related operations are all encapsulated in the system abstraction layer. For example, the sys_time_wait() in Figure 4 first consults the global context table and then invokes OS model utilities (timeWait()) with annotated metrics in a corresponding execution context.

In addition, the application context can be used for multiple other purposes. As described in Section 4.1.1, for the correctness of functional emulation, it encapsulates all global variables of the original application and external libraries in order to make it possible for them to be instantiated multiple times. Furthermore, the application context can also be used to record performance metrics.
of each individual simulated thread, such as dynamic basic block counts, to further enable the integration of any online prediction models.

Dynamic task creation under such context-aware host-compiled simulation needs to be carefully handled. Firstly, a dynamic process creation mechanism is required from the OS model all the way to the underlying simulation kernel, which serves as the foundation for creating application task during simulation. Secondly, a sequence of well-defined steps needs to be established to properly interact with the global context table and correctly create or inherit related simulation context.

We rely on the `sc_core::sc_spawn` SystemC API and an OS task wrapper, `os_task_wrapper`, to provide a generic, canonical OS adapter API `sys_thread_new` for dynamic task creation, as shown in the code snippet in Figure 5. 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 (`app_task`).

In order to correctly construct and inherit simulation contexts, task creation follows the specific steps highlighted in Figure 5: (1) the parent simulated task (`top_process`) first invokes the OS adapter function (`sys_thread_new`), and (2) inside `sys_thread_new`, the current simulation context is retrieved through the parent task’s simulation process handler (`Process 0`). Using the dispatched OS model pointer within the simulation context, (3) a new OS model task is then created by invoking the OS model’s task creation API (`taskCreate`). Then, (4) all the simulation context information is passed to the OS task wrapper function (`os_task_wrapper`), which is executed in a new SystemC process (`Process 6`) dynamically created by `sc_core::sc_spawn`. Finally, (5) the wrapper function registers itself with the global context table by adding its inherited simulation context indexed by the newly created SystemC process handler.

As such, during execution, only a top level OS task model needs to be instantiated once as a startup process, which constructs the corresponding application context. All the other application tasks can be dynamically forked using our context-aware API, where the simulation contexts required by the host-compiled simulator are automatically and internally inherited following the above-mentioned steps. In this way, any well-written external library can be easily mounted onto the OS model to emulate a realistic multi-threaded execution scenario.
We include a state-based power model to estimate the system energy consumption. The CPU idle/busy status and execution times are recorded by the OS model. During simulation, energy consumption is then calculated based on the corresponding power levels for various CPU states (number of active cores) and simulated application execution durations. Note that our proposed framework is developed as a general simulation platform, where various types of source-level estimation models with different levels of modeling detail and accuracy can be integrated [46].

4.2.2 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 [22], 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 6. During integration, we make use of our context-aware simulation technique to encapsulate the global variables from lwIP libraries and the OS/driver model contexts. 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.

We make use of the BSD socket interfaces in lwIP for networking programming in application tasks. These socket APIs are internally implemented by a set of lwIP Netconn APIs to create network connections and transfer data, as shown in the AppTask task model in Figure 6. 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 (nic_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 6. The incoming raw network packets are transferred from the nic_thread to the tcpip_thread through a tcpip mbox, and from there processed and further delivered to application tasks through the netconn recv mbox. 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 emulated through our proposed context-aware simulation technique. The lwIP OS adapter is modified to make use of the OS model adapter function `sys_thread_new` for dynamic task creation with consideration of the OS model simulation context. As such, the above-mentioned multi-threaded lwIP setup can be easily mounted onto the host-compiled simulator.

The lwIP Ethernet adapter includes all necessary low-level I/O functions to interface with the NIC hardware. With pointers provided by the application 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 simulated 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.

4.3 Network Simulation Model

We employ OMNeT++ together with the INET package [41] 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.

To model the energy consumption of network interfaces and radios, we utilize state-based energy consumption models from the INET library. Specifically, the transceiver power level for different operational states, such as receiving, transmitting or idle, is configured according to the modeled target specification. During the OMNeT++ simulation, the NIC energy consumption is then estimated over time based on the transceiver operational states and durations.

5 NOS DESIGN SPACE EXPLORATION

Due to the extremely large and heterogeneous NoS design spaces, we introduce a genetic algorithm-based framework to automatically perform NoS design space exploration. The overall setup of our exploration framework is shown in Figure 7. Each single NoS configuration is encoded into one genome, while the proposed host-compiled network/system co-simulation method is used for fast fitness assignment of each individual in the genome population. The whole process leverages standard genetic algorithm operators for selection, crossover and mutation to evolve the population for a threshold number of generations or until no further improvement can be achieved. In our work, we use the NSGA-II algorithm from [10]. The key challenge is thereby to encode NoS configuration parameters into genomes and automatically translate a certain genome into a simulation instance,
while seamlessly integrating the fitness assignment step into the overall algorithm flow. In order to achieve that, we develop model generators for automatic NoSSim simulation model instantiation. For exploration, design parameters from different design levels are encoded into separate segments in the integer genomes, as shown in Figure 7. Specifically, the application and system configurations for each device node are encoded separately into individual genes, which can be configured and tuned independently during exploration. The network-level parameters, such as node numbers and communication protocols, are globally encoded for the entire cluster. During fitness assignment, each individual genome in the explored population is then first translated into a set of JSON configuration files including parameters from applications, system devices and communication networks. After this translation stage, corresponding model generators will load the design parameters from JSON files and generate simulation model configuration files. To update or add new exploration parameters within our framework, a corresponding genome encoding needs to be defined in the genetic algorithm engine, the Genome2JSON converter needs to be modified to translate genes into JSON objects, and key-value pairs in JSON files need to be registered with model generators for translation into C/C++ header file and OMNeT++ configuration script entries. Examples of concrete JSON configuration files can be found in [44].

The application model is a C/C++ implementation of the application specification with desired partitioning. Reconfigurable task partitioning is implemented by inserting corresponding C/C++ macro definitions. The system model is a host-compiled SystemC simulation model with porting interfaces for application and network level models as described in Section 4, where available system-level configurations are also controlled by C/C++ macro definitions. Finally, the network model is an instantiation of an OMNeT++/INET simulation model with SystemC wrappers for each system simulation instance, where different MAC/PHY layer configurations, network topologies and number of node instances are generated by modifying OMNeT++ configuration scripts.

Finally, the application, system device and network models are compiled and linked together by a model composer to produce the final NoS simulation model. In the process, application models are back-annotated with corresponding timing and synchronization APIs provided by the host-compiled system simulator, while the system simulators are further inserted into simulation wrappers in the network simulation backplane. The generated NoS simulation model is executed
to collect simulated results. Estimated metrics from the network/system co-simulation are stored as the fitness values of the corresponding genome, which are then further used by the evolutionary search algorithm during its Pareto ranking and selection step.

Table 2 summarizes the exploration scope of our proposed framework. In general, application-specific parameters and any reconfigurable components in the underlying system and network simulation platforms can be explored, ranging from simple WSN setups over collaborative computing scenarios to complex edge computing cases represented by ECG, Vision Graph and DeepThings application case studies, respectively, that will be detailed in the following Section.

6 APPLICATION CASE STUDIES

Three representative case studies are introduced to evaluate NoSSim and demonstrate the overall network-level design and NoS exploration flow. We have released all application case studies in open-source form at [43, 45].

We first evaluate NoSSim and our automated exploration flow on a framework for distributed deep learning inference on resource-constrained IoT edge clusters, called DeepThings [47]. DeepThings uses a middleware and runtime system to dynamically distribute tasks under varying data and device availability. This results in complex parameter interactions and non-obvious design tradeoffs. We further employ two IoT applications from smart camera and healthcare domains using a traditional client-server computing paradigm [48] where different computation/communication ratios exist, 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 studies.

6.1 DeepThings

In IoT and edge computing scenarios, there is an increasing trend to process large-scale data directly on resource-constrained edge devices near the source. Within this context, various approaches have been proposed to leverage networked and distributed edge clusters to specifically perform DNN/CNN inference tasks collaboratively. We use the DeepThings framework from [47] for such a case study. DeepThings focuses on resource-optimized partitioning and dynamic distribution of early stage convolutional layers onto IoT edge clusters, where later-stage layers are offloaded to a gateway device.

An overview of DeepThings is shown in Figure 8. It includes an offline CNN partitioning step and an online middleware and runtime system to enable distributed adaptive CNN inference under dynamic IoT application environments. Before execution, DeepThings takes structural parameters of the original CNN model as input and feeds them into a Fused Tile Partitioning (FTP), as highlighted on the right in Figure 8. In FTP, memory- and communication-aware grid partitioning and layer fusion methods are applied to produce independently distributable tasks. In this process, different FTP partitioning parameters, such as grid dimensions and fusing depth, will result in varying computation and communication loads. FTP parameters together with model weights are then downloaded into each edge device. For inference, a DeepThings middleware is instantiated in each
IoT device to manage task computation, distribution and data communication. Its Data Frame Partitioner will partition any incoming data frames from local data sources into distributable and lightweight inference tasks according to the pre-computed FTP parameters. The Runtime System in turn loads the pre-trained weights and invokes an externally linked CNN inference engine to process the partitioned inference tasks. In the process, the Runtime System will register itself with the gateway device, which centrally monitors and coordinates work distribution. The DeepThings middleware uses a work stealing approach for low-overhead dynamic load balancing. If its task queue runs empty, an IoT edge node will poll the gateway for devices with active work items and start stealing tasks by directly communicating with other DeepThings runtimes in a peer-to-peer fashion. Finally, after edge devices have finished processing all tasks for a given data source associated with one of the devices, the gateway will collect and merge partition results from different devices and finish the remaining offloaded inference layers.

A key aspect of DeepThings is that it is designed to be independent of and general in supporting arbitrary pre-trained models and external inference engines. We implemented DeepThings in C/C++, with network communication modules in runtime systems realized using TCP/IP with socket APIs. Without loss of generality, we instantiate DeepThings using the You Only Look Once (YOLOv2) object detection framework [34] with its C-based Darknet neural network library as external CNN inference engine. We apply partitioning and distribution to the first 16 layers in YOLOv2 (12 convolutional and 4 maxpool layers). The remaining layers execute on the gateway.

6.2 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 [11]. This type of relationship is defined as vision graph [8]. 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.

We use the formulation of different vision graph discovery scenarios from [48], which is summarized in Figure 9a. Computation is divided into four stages (image capture, keypoint extraction, feature matching and graph discovery), along which we define three offloading levels and corresponding task mappings. For all applications in our studies, one stage 0 task is always associated with each client device based on the availability of necessary physical sensors. In case of the vision graph, a single stage 3 task is further assigned to the server as it requires image matching information from the entire camera network. Note that feature extraction requires image information from

---

Fig. 8. Overview of the DeepThings framework.
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 in the underlying communication network topology, and the server is responsible for matching all other possible image pairs.

We have implemented the vision graph application using the ezSIFT [42] library. Video sequences from the Freiburg-Berkeley Motion Segmentation Dataset [29] are used as our input data.

6.3 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 [36], where ECG signals are used to detect heart arrhythmia, i.e. irregularly fast or slow heart beats, which may lead to strokes or heart failure.

For our case study, we follow the setup from [48] as shown in Figure 9b. The detection flow is divided into four stages (filtering and heartbeat detection, heartbeat segmentation, feature extraction and diagnosis) with four possible offloading levels. Different from vision graph discovery, source signals and analyses in each device are independent from each other in ECG diagnosis. Thus, the stages after filtering can be freely mapped onto client or server nodes.

The ECG processing flow is implemented in C and instrumented and back-annotated with target platform metrics and OS model APIs for system integration. ECG data records from the MIT-BIH Arrhythmia Database [25] are used as input data.

7 EXPERIMENTAL RESULTS

For our experiments, we set up different NoS configurations in a wireless local area network (WLAN) topology. Two types of devices are used as reference platforms for profiling, back-annotation and simulator validation, including a Raspberry Pi 3 model B (Pi3), which is a 1.2 GHz ARM Cortex-A53 quad-core device, and a Raspberry Pi 0 (Pi0), which is a 1 GHz ARM11 single-core device. Each device is running Raspbian, a Linux-based operating system for Raspberry Pi. To extract timing information from the physical Raspberry Pi devices, we record cycle counters using PAPI, and we use the performance metrics obtained from the Raspberry Pi devices to back-annotate both the application and network stack source code. Network-related timing information is provided by the INET/OMNeT++ library. For energy information, we use state-based power models to estimate processor and radio interface energy consumption, where nominal data from data-sheets of Pi0,
Pi3 and commercial network interface cards are used for back-annotation. All simulations and explorations are performed on a quad-core Intel i7 workstation running at 2.6 GHz.

7.1 Exploration Setup
We have implemented our NoS simulator (NoSSim) on top of SystemC and OMNeT++ and released it in open-source form at [45]. The final NoS simulation setup used for experiments is shown in Figure 10. 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.

We have implemented model generation tools and the overall exploration engine in Python, utilizing the NSGA-II library/implementation from DEAP [16] as the genetic searching method to simultaneously optimize several design objectives. Our NoS exploration tool has been released in open-source form at [44]. The genetic algorithm configuration used for our experiments is shown in Table 3. In our case studies, individuals in the initial generation are randomly generated from their corresponding value sets, and further evolved within the valid ranges. We aggressively pick large crossover and mutation rates to diversify the Pareto front. In all applications, we are able to observe a convergence on the optimization results within the maximum number of generations.

7.2 Simulation Speed and Accuracy
We first evaluate NoS simulation speed and accuracy using DeepThings, where a Pi3 cluster of up to eight edge and a single gateway device is instantiated in NoSSim and compared against a real-world setup. The general NoSSim setup follows Figure 10. In this section, we make use of...
single-core edge and quad-core gateway system configurations in both real-world and simulation setups. For the application tasks, we instantiate corresponding tasks from DeepThings gateway and client services. Additionally, we obtain real-world execution time measurements from a Pi3 device cluster to back-annotate the application and lwIP source code in the NoSSim setup. We also calibrate the configurable parameters in the lwIP stack against the Linux-internal TCP/IP stack to achieve similar communication throughput. For the MACPHY layer, we use the IEEE 802.11n communication protocol in NoSSim simulations, which is the same as in our Raspberry Pi cluster.

7.2.1 Simulation Speed. Figure 11 shows the simulation wall-clock time and speed for NoSSim running DeepThings with different edge node numbers and partitioning dimensions. Wall-clock time is shown as the average simulation runtime per input data frame, where the wall-clock time for a purely non-functional timing simulation and for functionality emulation are plotted separately and stacked together. In general, the simulation wall-time increases linearly with more edge nodes introduced to the edge cluster, which is caused by extra synchronization and context switch overhead. We can observe that compared to a timing-only simulation, functional simulation overhead can contribute more than 40% to the total simulation duration when only one edge node device is simulated. However, as more edge nodes are introduced, overhead of simulating communication can contribute more than 40% to the total simulation duration when only one edge node device is simulated. However, as more edge nodes are introduced, overhead of simulating communication becomes dominating. In terms of FTP partitioning parameters, a finer granularity will expose a larger amount of communication data and thus require relatively more OMNeT++ and SystemC communication and synchronization events to be simulated. As a result, larger FTP partitioning dimensions under the same cluster size will always result in longer simulation duration.

The average simulation speed for simulations with and without functional emulation are 0.23 and 0.35 simulated seconds per wall-clock second, respectively. Major differences in simulation speed occur with smaller numbers of edge nodes, where the application-level computation duration is the dominating factor for simulation runtime. However, application-level parameters of DeepThings do not have significant effects on simulation speed. From a more general perspective, a coarser annotation granularity and relatively longer computational code can result in a smaller amount of simulated events and can thus potentially result in more simulated seconds within the same amount of wall-clock time.

7.2.2 Simulation Accuracy. Two different input data stream scenarios are used to verify the accuracy of our simulator. The execution latencies of one data frame are compared between NoS simulation and real-world execution. The mean absolute percentage error is measured and demonstrated. As shown in Figure 12a, we first fixed the number of data sources to be one, and change the IoT edge cluster size and FTP partitioning dimensions. The total number of computing nodes increases from 1 to 6, where FTP partitioning dimensions of 3x3 and 5x5 are applied in each case. Results show that the average error across all parameter setups is 16%, while the maximum error can be as large as 28%. Nevertheless, for the purposes of exploration, fidelity, i.e. relative accuracy, is
most important, where parameter sensitivities are captured with high fidelity by our framework. A similar scaling of inference latency with an increasing number of devices is demonstrated in simulation as compared to the real Pi3 cluster, where inference speedups gradually slow down after 4 devices. In terms of FTP partitioning dimensions, because of the smaller amount of overlapped data and computation/communication redundancy, larger partitioning granularities always result in better performance on the Raspberry Pi3 cluster, where our simulation demonstrates the same relative differences.

We further explore a scenario with multiple data sources, where the cluster size is fixed to be 6, and the number of data sources increases from 1 to 6. As shown in Figure 12b, the average accuracy for multiple data sources is 87% with a 23% maximum error, and the scaling trends and FTP parameter sensitivities are also correctly captured. When more data sources appear, the stealing activities are adaptively reduced and the longest inference latency converges towards the single device execution performance in both simulation and real-world execution.

In both scenarios, larger percentage errors occur when more idle devices are available and more stealing activities are performed. NoSSim simulation errors can come from both system and network simulation. In this particular case, however, the major factor is the difference in behaviors between the lwIP and Linux TCP/IP stacks. In DeepThings, a lot of small packets are exchanged for transferring the necessary metadata of partitioned inferences, such as data source ID, reuse data status or partition ID. Such a large number of small packets trigger TCP delayed acknowledgments (ACKs), where several ACK responses are combined into a single response and the combined acknowledgement packets can be delayed and rescheduled as controlled by stack-internal cyclic timers. In lwIP, in order to keep the implementation lightweight, several TCP events are lumped together within a fast TCP re-transmission timer with a period of 250 ms. In Linux TCP/IP stack, by contrast, various timers are defined with finer granularity to handle different TCP events, where the delayed ACK is transmitted every 40 ms. As a result, NoSSim always underestimates the performance under a larger amount of stealing activities. Nevertheless, the proposed network/system co-simulation framework is accurate enough to correctly capture parameter sensitivities with high fidelity and thus serve as a fast estimation tool in our automated design space exploration framework.

## 7.3 Design Space Exploration Case Studies

We use vision graph discovery and ECG arrhythmia detection to demonstrate exploration of design-time parameters such as per-client offloading levels, system and network configurations optimized for both throughput and energy consumption. Exploration parameters for both applications are shown in Table 4. Finally, we perform a comprehensive design space exploration for DeepThings. In addition to design-time parameters from different design levels, its middleware and runtime systems are also deployed in NoSSim to emulate and evaluate dynamic, adaptive task distribution.
Table 4. Vision graph discovery and ECG diagnosis exploration parameters.

<table>
<thead>
<tr>
<th>Offloading Levels</th>
<th>Vision Graph Discovery</th>
<th>ECG Arrhythmia Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Types</td>
<td>O-0, O-1, O-2</td>
<td>O-0, O-1, O-2, O-3</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>Client (Data Source): 6, Server: 1</td>
<td></td>
</tr>
<tr>
<td>Communication Protocol</td>
<td>802.11b@1Mbps, 802.11g@6Mbps, 6LoWPAN</td>
<td></td>
</tr>
</tbody>
</table>

Table 5. DeepThings exploration parameters.

<table>
<thead>
<tr>
<th>FTP Grid Dimensions</th>
<th>3x3, 4x4, 5x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fusing Layers</td>
<td>4, 8, 12, 16</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>Edge: {1, 2, 4}, Gateway: 4</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>Edge: 1~12, Gateway: 1</td>
</tr>
<tr>
<td>Communication Protocol</td>
<td>802.11b@{5.5, 11Mbps}, 802.11g@{6, 18, 48, 54Mbps}, 802.11n@600Mbps</td>
</tr>
</tbody>
</table>

(a) Latency optimization progress. (b) Energy optimization progress. (c) Pareto front size per generation.

Fig. 13. Genetic algorithm exploration progress.

behaviors effects on inference latency, per-device inference energy consumption and memory usage. A summary of DeepThings exploration parameters is shown in Table 5.

7.3.1 Convergence and Diversity. Figure 13a and Figure 13b demonstrate the representative optimization progress during exploration for normalized processing latency and average edge device energy consumption, respectively. In each generation, the minimum latency and energy consumption among the Pareto-optimal solutions are plotted in the graph.

In vision graph discovery and ECG diagnosis applications, offloading levels and task mapping are statically fixed for each data source device, and all the intermediate or final results are centrally collected in the server/gateway device. As a result, communication protocols and offloading levels are the key factors affecting design metrics. Note that the offloading levels and device types can be different for each individual device, which further introduces a large design space and non-trivial parameter interactions. For vision graph discovery, the minimal latency is slightly improved in the first 10 generations and sharply reduced after the 25th generation. The reason for this trend is that there is one particular design configuration (minimum latency configuration in Table 6) that can significantly reduce the processing latency compared to other solutions, which is hard to be discovered during the evolutionary search. Still, our framework successfully discovers such a design candidate and achieves identical optimization results to the manually explored solutions in [48]. Per-device energy consumption in vision graph discovery has a relatively smooth parameter sensitivity and is quickly optimized and converged in the first 6 generations. By contrast, ECG diagnosis has a smaller amount of computation, and communication is the major latency bottleneck. Thus, exploration is mainly focused on tuning offloading levels to reduce the data transmission...
overhead. With a smaller latency, the idle energy consumption will also be reduced, resulting in better energy efficiency. As a result, parameter sensitivities of ECG diagnosis are relatively straightforward, and both latency and energy converge quickly within the first 10 generations.

For DeepThings, application task mappings are dynamically adjusted by the work-stealing middleware. As a result, parameters from different design levels have more complicated sensitivities and need to be explored all together. During exploration, inference latency is improved from 6.76s to 4.77s with a 29% reduction. To optimize collaborative inference performance, the edge cluster size and individual device architecture need to be carefully configured considering available communication bandwidth. For both energy and latency, non-obvious interactions between communication and computation exist. Any straightforward homogeneous cluster setup can not efficiently explore such interactions, and a large number of possible heterogeneous cluster configurations need to be considered during exploration. Additionally, different FTP parameters and the dynamic, adaptive middleware also introduce complicated parameter sensitivities. All combined, complex and large design spaces can result in a long exploration process before convergence. Note that memory usage is also optimized through the exploration process. However, device memory usage only depends on the FTP partitioning dimension and fusing depth, which have only 12 possibilities. As a result, with a population of 80 individuals, the minimum memory design (9MB memory with FTP-5x5, 4 fused layers) can be easily discovered from the first evolutionary generation. Nevertheless, the memory usage is still carried along during the exploration process as an optimization target for Pareto-optimal solution selection.

Figure 13c further shows the size of Pareto front along with the algorithm progress. The vision graph discovery and ECG diagnosis applications have relatively simple design spaces with final Pareto fronts consisting of only 5 and 2 non-dominated solutions, respectively. During the exploration process, as better design candidates are discovered, the Pareto front sizes fluctuate in both cases. For DeepThings, more parameters and design metrics introduce a complicated and larger Pareto-optimal front, which is gradually discovered during the exploration process. The number of non-dominated solutions increases from 18 to 71 after 30 generations, where the corresponding optimization metrics improve alongside as shown before. As can be seen in this case study, our design space exploration framework is able to maintain a good diversity for the Pareto front to accommodate different design constraints and tradeoffs across various NoS application scenarios.

To accelerate the exploration runtime, we utilize a parallel setup with 6 processes on our workstation to evaluate each generation. The average wall-clock time for evaluating each generation of vision graph discovery and ECG diagnosis is 4.7 and 4.5 minutes, while the total exploration times with 30 generations are 2.2 and 2.3 hours, respectively. For the DeepThings case study, the evaluation time is around 80 minutes per generation, while the entire design space exploration takes around 40 hours to complete. Note that for DeepThings, only timing-related events are simulated to speed up the fitness assignment procedure.

7.3.2 Exploration Results. The energy vs. latency design space for vision graph discovery and ECG examples is shown in Fig. 14. In the graphs, colors/shades indicate offloading level choices, icon shapes represent different client types, and design points with specific protocols are annotated. For heterogeneous client cluster configurations, we classify them based on the majority offloading levels and device types. Specifically, lighter shades indicate a smaller proportion of the majority offloading level within the client cluster, and hollow vs. filled markers represent heterogeneous vs. homogeneous cluster device configurations, respectively. The minimum latency and energy consumption design points are further highlighted and their configurations are summarized in Table 6. We measure latency as 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, which is mainly affected by application offloading levels and communication protocols. As can be seen in Figure 14, the communication protocol is the dominating factor for latency, while the offloading level will further distinguish latency within each protocol cluster. Client energy consumption per data frame for both case studies includes both idle and busy device energy consumption. As such, energy consumption is tightly correlated with latency, where longer latency will result in either larger busy or idle energy consumption. However, different from application latency results, which are mainly constrained by communication bandwidth, changing device types can also significantly affect the energy consumption of edge devices.

Offloading level O-1 in vision graph discovery can achieve the best parallelism among edge devices with minimal amount of communication overhead. As a result, the design solution with minimal latency uses such an offloading level for all clients. For per-client energy consumption, however, trade-offs do exist. Higher offloading levels can map more tasks to the server device and reduce client energy consumption, while a medium offloading level (O-1) with shorter overall latency will have smaller idle energy consumption per client. As such, the solution with minimal per client energy consumption has mixed offloading levels among different clients, where one client has O-2 and others have O-1. In terms of system configuration, Pi3 is always preferred for better performance. Counter-intuitively, however, a Pi3 can also achieve a better energy efficiency. This is because the reduction of idle energy consumption with a Pi3 client outweighs the reduction of busy energy consumption in Pi0. For communication configurations, 802.11g has a better energy efficiency and larger communication throughput compared to 802.11b. For 6LoWPAN, energy consumption of the network interface card is generally lower compared to 802.11 protocols. However, since the vision graph application generates a large amount of data, 802.11g can have a lower radio energy consumption as well. This is because IEEE 802.15.4 PHYs only support frames of up to 127 bytes, where large data chunks need to be sliced into small packets and more overhead for control headers is incurred, resulting in a higher energy consumption for transferring the application data.

Different from the vision graph case, the computation workload in ECG diagnosis is generally small and communication overhead is more dominant. As such, network protocols can significantly affect overall output delay and 802.11g is always employed. Changing the client device type from
Pi0 to Pi3, however, will not result in any significant performance benefits, and Pi0 is generally preferred for better energy efficiency. For application-level parameters, Offloading level O-0 is preferred for optimizing both latency and energy metrics, which maps most of the computation workload locally in client devices and require minimal amount of data transmission. Because of all such relatively straightforward parameter sensitivities, the final Pareto front only consists of the minimal-latency (blue) and minimal-energy (red) design points.

Finally, Figure 15 shows the DeepThings design space. Figure 15a shows the design points explored in the energy vs. latency space, where the corresponding Pareto front is highlighted.

The corresponding design points with minimum latency and energy consumption are highlighted in Figure 15a and summarized in Table 7. Interestingly, the dynamic interactions between communication and computation result in non-obvious design trade-offs. The computing cluster configuration providing fastest inference does not only consist of quad-core devices, but of a heterogeneous collection of edge nodes. This is because the relatively slow dual-core devices will spend more time on task processing and have fewer stealing activities. In this way, the reduced data transmission efforts actually result in better performance, where more tasks are quickly processed in the original data source instead of being handed out. Furthermore, the fastest configuration only uses a total of 5 devices, as communication overheads outweigh the benefits of adding additional compute resources to the cluster.

Similar non-obvious trade-offs also exist for the minimum energy design point. In general, as more edge devices are introduced, the inference workload is more widely distributed, where each device may get fewer processing tasks and thus result in a lower per-device energy consumption. In addition, although dual-core devices provide better energy efficiency, the energy consumption from the network interface card is not negligible when more edge devices are introduced into the cluster. As a result, using a quad-core device at the data source results in a smaller amount of task distribution and lower NIC usage, which in turn compensates the relatively higher quad-core energy consumption for task processing.
Table 8. DeepThings parameter trade-offs.

<table>
<thead>
<tr>
<th>Design Metrics</th>
<th>Device</th>
<th>Cluster Size</th>
<th>Grid Dimension</th>
<th>Fusing Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>Powerful</td>
<td>Medium</td>
<td>Coarse</td>
<td>Medium</td>
</tr>
<tr>
<td>Energy</td>
<td>Medium</td>
<td>Large</td>
<td>Coarse</td>
<td>Small</td>
</tr>
<tr>
<td>Memory</td>
<td>-</td>
<td>Fine</td>
<td>-</td>
<td>Small</td>
</tr>
</tbody>
</table>

Finally, Figure 15b shows the overall DeepThings design space for latency, per-device energy and per-device memory consumption, where the design point with minimum memory consumption is highlighted. As can be seen, the best design candidate for each individual optimization target comes with different configurations, while a single NoS configuration that can simultaneously optimize all metrics does not exist. The three optimization targets in the DeepThings application have different requirements on design parameters, as summarized in Table 8. Quad-core devices may result in better performance, while dual-core devices provide better energy efficiency. The choice of a cluster size, however, largely depends on the communication bandwidth and overhead. In our case, best performance is achieved on a 5-device cluster, while minimum energy consumption is obtained on a full-size (12-device) cluster. Note that for MAC/PHY layer parameters, 802.11n is always preferred during the exploration. Under the large volume of DeepThings transmission data, 802.11n demonstrates better throughput, latency and energy efficiency. Additionally, application parameters can also significantly affect the design metrics. In DeepThings, a coarse partitioning dimension is always preferred for better latency and energy results, where smaller computation and communication redundancies exist because of less overlapped boundary data. A smaller FTP fusing depth will have more computation offloaded to the gateway device and thus result in lower edge energy consumption, while a medium fusing depth (12 layers) can achieve the best tradeoff between computation parallelism and communication overhead, providing shorter inference latency. The FTP parameters also decide the per-device memory consumption. Finer partitioning grid dimension and smaller fusing depth will result in smaller memory footprint, which is independent of system/network configurations.

As can be seen, such complicated and application-dependent parameter interactions in DeepThings are hard to be captured and explored manually. By applying our network-level design flow, non-obvious design trade-offs are automatically discovered and optimized. Comparing against a single edge device execution (FTP-3x3, 16 fused layers), our network-level design flow discovers design candidates providing more than 90% per-device energy consumption reduction (Table 7) and a 3.6x improved inference speedup (Table 7) on a resource-constrained IoT edge cluster. Comparing against the setup with minimum latency in [47], where a FTP-3x3 with 16 fused layers is applied on an edge cluster consisting of 6 single-core devices, latency can be further reduced by more than 45% in the heterogeneous cluster setup (Table 7) generated from exploration.

8 SUMMARY AND CONCLUSIONS

In this paper, we presented a comprehensive and fully automated framework for network-level design space exploration of resource-constrained networks-of-systems (NoS). A fast and accurate NoS simulator (NoSSim) is first introduced. It combines a flexible host-compiled system simulation environment with a configurable network simulation backplane, achieving more than 86% accuracy on average with high relatively fidelity compared to a real world edge device cluster while running at an average simulation speed of 0.29 simulated seconds per wall-clock second. Based on NoSSim, an automated NoS design space exploration framework is then developed, where NoSSim for fitness evaluation is combined multi-objective genetic algorithm for automatic NoS design parameters optimization under different design objectives. We apply our network-level design flow on several comprehensive case studies. Results indicate that our framework is able to rapidly explore
complicated design spaces and provide better solution than manually optimized designs. We have released NoSSim, our NoS design space exploration framework and application case studies in open-source form at [44, 45]. Future works can include investigation of analytic models for NoS fitness evaluation as well as development of dedicated, problem-specific optimization heuristics to replace generic, meta-heuristics such as genetic algorithms.

ACKNOWLEDGMENTS

This work was partially supported by National Science Foundation (NSF) grant CCF-1763848.

REFERENCES


