Platform Choices and Design Demands for IoT Platforms: Cost, Power and Performance Tradeoffs

Deming Chen[1,3], Jason Cong[2], Swathi Gurumani[3], Wen-mei Hwu[1], Kyle Rupnow[3,*], Zhiru Zhang[4]

[1]Electrical and Computer Engineering Department, University of Illinois Urbana-Champaign, USA
[2]Electrical and Computer Engineering Department, University of California Los Angeles, USA
[3]Advanced Digital Sciences Center, Singapore
[4]School of Electrical and Computer Engineering, Cornell University, USA
[*]k.rupnow@adsc.com.sg

Abstract: The rise of the Internet of Things has led to an explosion of new sensor computing platforms. The complexity and application domains of IoT devices range from simple self-monitoring devices in vending machines to complex interactive devices with artificial intelligence in smart vehicles and drones. As IoT developers wish to meet more aggressive platform objectives and protect market share through feature differentiation, they must choose between low-cost, and low-performance CPU-based commercial-off-the-shelf (COTS) systems, and high-performance custom platforms with hardware accelerators such as GPU and FPGA. Both COTS and custom platform designs introduce a variety of design challenges — the extreme pressures on time-to-market, design cost, and development risk are also driving a voracious demand for new CAD technologies to enable rapid, low cost design of effective IoT platforms with smaller design teams and lower risk.

In this article, we present a generic IoT device design flow and discuss platform choices available for IoT devices to efficiently tradeoff cost, power, performance and volume constraints: CPU-based systems, and custom platforms that contain hardware accelerators such as FPGA and embedded GPUs. We demonstrate this design process through a driving application in computer vision. We also present current critical design automation needs for IoT development and demonstrate how our prior work in CAD for FPGAs and SoCs begin to address these needs.

1. Introduction

The Internet of Things is driving an explosion in sensor computing platforms in consumer, commercial, and industrial domains. IoT devices span a wide range of application domains including fitness trackers, drones, cameras, health monitors, and home automation for personal use, and smart grid, transportation, logistics, manufacturing and agriculture applications for commercial or industrial use. IoT devices are transformative – sensors, local computation, and integration with cloud computing moves intelligence to edge devices and allows global decision-making based on detailed, local sensor measurements. Through these applications, improved operational intelligence can improve safety, and efficiency as well as directly improve functionality.

Market study and analysis for IoT applications predict substantial growth in both device sales and cloud computing services. IoT application spaces are poised to become the largest electronics market for the semiconductor industry, and the wide range of application domains spans from simple applications that may need only an 8-bit microprocessor to high-end systems that may

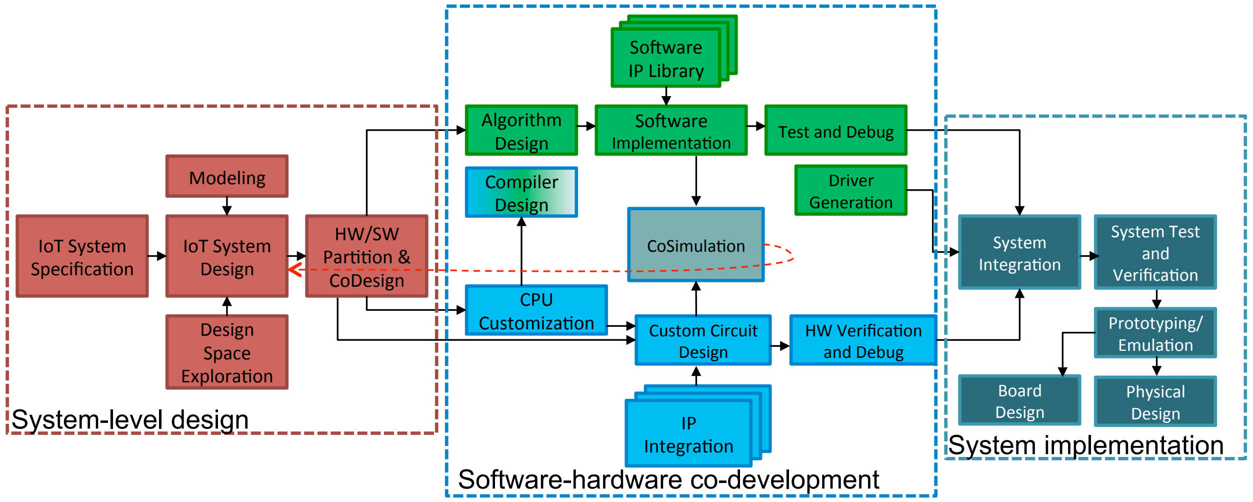

Fig. 1. *Typical IoT Device Design Flow*

need a combination of high-performance local computing with offloaded cloud computing. This wide variety in application complexity corresponds to multiple dimensions: IoT devices may be battery operated and power-constrained, or operating on wired power; devices may be performance constrained or latency (and performance) insensitive; devices may be size and weight constrained or have size flexibility in installation size and location; devices may be cost constrained to compete in consumer markets or (comparatively) cost-insensitive due to application value-add.

Although many standardized, low-performance and low-power IoT platforms exist [1, 2, 3, 4, 5], there is still a significant need for high-performance platforms in order to meet computational needs of data-intensive applications moving intelligence into edge devices. Generally, these high-performance platforms consist of graphics processors (GPUs) or fixed-function implementations in FPGA or ASIC implementations. These options improve performance and power consumption, but IoT applications must balance power, performance, size, and cost through the use of embedded GPUs [6, 7, 8, 9, 10] and FPGA-based SoCs [11, 12], which can achieve even better energy efficiency [13, 14, 15]. Recent advances in high-level synthesis [16, 17, 18, 19, 20, 21] have improved the ease of FPGA programming, which simplifies use of FPGAs in IoT applications.

To date, most IoT devices are based on existing platforms, primarily CPU-based platforms [1, 2, 3, 4, 5], but also including embedded GPU [22, 23], or FPGA [24] platforms. When application development identifies needs for custom computation, feature differentiation, and improved efficiency in computation latency, power/energy, or physical device size, application developers will need to meet these more aggressive platform objectives with custom platforms. However, custom platform designs introduce a variety of challenges in design and design cost; design automation plays a key role in making these designs feasible with lower development risk and costs. Despite the advantages of custom platforms, complex and challenging design flows remain a barrier to adoption of custom platforms, especially with integrated hardware accelerators. In the design of these platforms, computer aided design (CAD) plays an important role in supporting design space exploration and meeting design objectives while simultaneously reducing time-to-market, design cost, and development risk to ensure both technical and commercial success.

Designing custom platforms will thus be a required for many future IoT devices; custom platforms can be several orders of magnitude faster and more power/energy efficient than CPU-based alternatives [25]. These custom platforms can more effectively integrate security, privacy and reliability features, as well as application-specific acceleration for key feature differentiation. Feature differentiation can be critical for capturing and retaining market share, and custom platforms play an important role in preventing competitors from simply reproducing copies of the same IoT platform. Design pressure for custom platforms drives demand new design automation technologies, with critical needs in system-level integration, IP integration, and verification.

In this article, we present the generic IoT device design flow and discuss platform choices available for IoT devices to efficiently tradeoff cost, power, performance and volume constraints: CPU-based systems and custom platforms that contain hardware accelerators such as FPGA and embedded GPUs. We demonstrate this design process through a driving application in computer vision. As computer vision algorithms become mature, the application of advanced vision algorithms becomes a key driver for intelligence in edge devices. However, vision applications are highly data-intensive; the costs in latency, communications, and computation to move video data to centralized cloud computing would quickly become a limiting factor in the scalability of these vision solutions. Vision systems are increasingly expanding into IoT applications: analysts predict connected imaging equipment will grow by 17% annually to 2020 [26]. Computer vision IoT applications will require local computation in order to achieve the performance and scalability objectives for large scale deployments in body cameras, surveillance systems, building entry, and public transport systems.

Through the example of facial detection and recognition, we detail the design process and computer aided design challenges and describe the trends and demands for computer aided design for IoT systems. Starting with standardized CPU, GPU and FPGA platforms, we describe implementation platforms, examine current critical design automation needs for both COTS-based and custom platform devices. We discuss both current missing pieces for design automation as well as our ongoing work targeting these gaps; together, CAD promises to reduce design time, cost and risk to serve as the bridge that makes design and implementation of custom platforms for IoT devices effective and timely.

2. Vision-Based IoT Systems

Computer vision is a particularly compelling application domain for sensor computing systems. In general IoT sensor computing systems have analog sensors to monitor a local environment and either notify a remote server of conditions or locally perform decision making based on sensor values. Image sensors can provide detailed sensor information for complex decision making: drones use vision to detect and avoid obstacles during flight, factories monitor machine conditions, difficult to reach mechanical parts, and manufactured parts, highways monitor traffic congestion and detect unsafe or illegal behavior, surveillance systems can monitor playgrounds and swimming pools, monitor livestock in agriculture or search for people for access control systems or terrorism avoidance.

Scalability of vision applications is a key driving concern in terms of both initial cost as well as ongoing operations cost. Although a single camera with the vision software is valuable, the true value of the system is when many cameras can be deployed to monitor large areas and automatically detect and respond to or send alerts. With many cameras, the individual cost of each camera must remain reasonable, but the cost of operating many cameras must also be considered. Simple cameras that send large video streams to the cloud for computation require significant, ongoing costs in both bandwidth use for data upload as well as computation costs in the cloud. In contrast, localized computation can reduce upload bandwidth and cloud computation demand

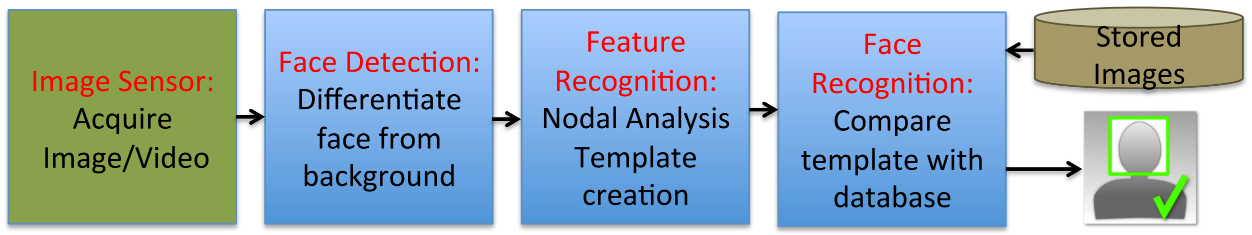
Image Sensor: Acquire Image/Video
Face Detection: Differentiate face from background
Feature Recognition: Nodal Analysis Template creation
Face Recognition: Compare template with database
Stored Images

Fig. 2. *Face Recognition Overview*

which improves scalability. Thus, vision IoT systems must balance between low-cost local systems with large cloud communication and compute demand and high(er) cost local systems that localize computation in order to reduce communication and compute requirements in the cloud.

2.1. Face Recognition

Face recognition is a key example of vision applications for IoT; recognizing individuals in surveillance footage is a key technology that enables many end-use applications including building access control, mass transit entry, and criminal suspect identification among others. As recognition algorithms mature, there is significant demand to integrate the recognition capability into edge-device cameras.

Face recognition systems start by identifying high quality video frames and detecting faces within the frame. Ideally, faces have consistent lighting, head position, size, and pose, but in practice systems must be trained to handle a variety of face orientations with varying picture quality. Every face has about 80 distinguishable landmarks called nodal points that comprise facial features. The recognition system translates nodal-point measurements into a faceprint representing the combined features that can be compared to faces in a database for identification. An overview of face recognition systems is shown in Figure 2.

Face recognition is emerging as a forensic tool of substantial importance. It is increasingly used in law enforcement in products such as body cameras to identify suspects in the line of duty. However, face recognition in such critical applications is time sensitive; for the recognition result to be useful, it must also be timely, and thus the system must be able to recognize subjects in real time. However, CPU implementations, particularly in embedded systems, do not achieve real time performance [27] even for face detection for resolutions higher than 480p and face recognition is significantly more complex than detection alone. Current commercial face recognition systems use high-end server class machines [28, 29, 30] coupled with hardware accelerators including high-performance GPUs to achieve real-time performance for face recognition systems. However, these high-end implementations cannot be directly used with body cameras: the form factor, power, and cost of these systems limit deployment.

Through local acceleration of the algorithm and partitioning of the algorithm between local and cloud compute resources, an IoT system that meets cost, size, and performance goals can be deployed. In the following sections we will describe the process of prototyping the software system and specifying local platform characteristics in order to select a CPU, GPU or FPGA-based local platform. Then, we will discuss CAD demands for platform design of a custom platform that targets cost, power, performance, and size constraints.

3. IoT Device Design Flow

IoT device design consists of software development for CPU implementation, interface drivers, and, when application demands dictate, hardware design for custom accelerators, CPU customization, and board-designs. In order to minimize time-to-market, software and hardware are often developed in parallel, with a software team concentrating on software features, embedded compilation, device drivers and integration with cloud computation services. In parallel, the hardware team performs system level modeling, component selection, design implementation, integration and verification. Despite the generally parallel development processes, the software and hardware design flows influence each other; software algorithm demands may alter hardware performance objectives, and hardware implementation choices can influence how software is designed and implemented. Indeed, if a hardware implementation is created, there may be no need for design and optimization of the software version.

Many IoT applications begin with prototyping entirely in software on existing embedded platforms. To minimize design and development costs, application providers prefer to reuse existing platforms if possible, so initial prototypes are typically designed on existing CPU-based platforms. With the initial implementation, designers can evaluate performance and quality to determine whether a CPU-only, GPU-based or FPGA-based platform is necessary to achieve goals. Many simple applications may meet all cost, performance and power goals when implemented on existing platforms, yet applications such as vision systems typically need higher performance and thus either GPU or FPGA-based systems in order to meet design objectives.

In this article, we primarily concentrate on design automation for the hardware portion of a device design flow, and thus we assume that the application has already identified a need for a customized platform whether the platform is a customized CPU, a GPU-based platform, FPGA-based platform or a hybrid offering multiple computation devices. An overview of a typical IoT device design flow is shown in Figure 1. At a high-level, the design flow consists of three phases: system-level design, software-hardware co-development and system integration and implementation. Though the three phases generally happen in a feed-forward manner, a feedback path exists between the cosimulation and system design phase to evaluate and regenerate a system design and software hardware partitioning that can meet design objectives. There are key challenges in the design cycle and thus key needs for design automation in modeling (Sections 5.1, 6.1.2, 6.2.1), device driver generation (4.1), CPU customization (6.1), circuit design (6.2.2), verification and debug (6.2.3), prototyping (4.2) and IP and system integration (6.2.2) in this paper. The design flow in Figure 1 details the steps required across typical IoT platforms, but certain steps of the flow may not be applicable depending on the target platform. For example, circuit design steps are not applicable to COTS-based IoT flow.

Although we primarily concentrate on design automation for the hardware portion of the device design flow, automation also plays a critical role in software design processes, and integration of hardware design automation with analysis of the embedded software together with automated creation of embedded compilers (for customized CPUs), and device drivers (for integrating standard analog/RF/digital accelerator components) is critical so that software design can effectively use the hardware platform, and the hardware design can effectively evaluate platform objectives based on up-to-date software needs. We will discuss these automations in further detail in Section 5.

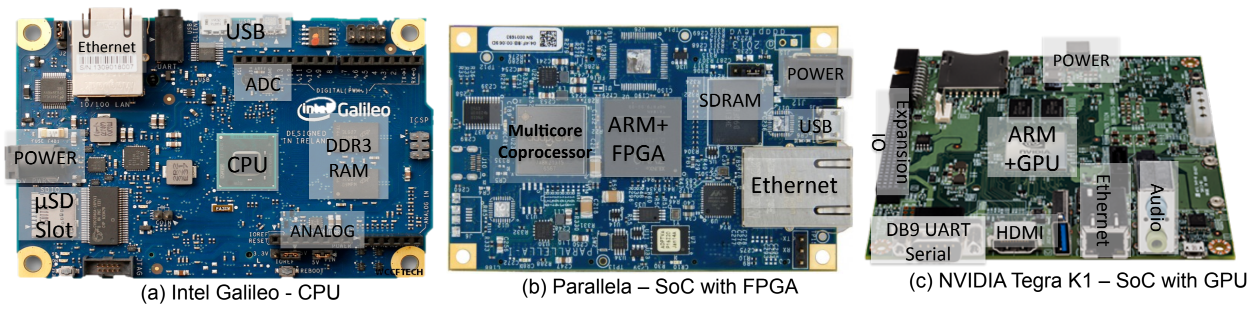

Fig. 3. *Example IoT Development Boards*

4. IoT Prototyping Platforms

Development typically begins with application and device prototyping. As discussed above, the first prototypes assist in identifying platform computation needs, and the feasibility of meeting quality goals within the performance, power, and size limitations of existing platforms. After this pre-prototyping, a designer has specified the platform characteristics of the desired target, and can now begin by prototyping the solution on a platform that more closely matches the expected final platform characteristics.

As discussed above, in this article, we assume that the initial prototyping identified a need for some platform customization. Thus, a first step for IoT development for custom platforms is to begin prototyping using a platform or combination of platforms that matches the desired target as closely as possible. This may be an existing CPU platform [1, 2, 3, 4, 5], a GPU platform such as the Tegra K1 or X1 [7, 6], or FPGA platforms such as the Xilinx Zynq-based Parallela [24] as shown in Figure 3. These platforms contain standard communications, sensor interfaces, and general purpose I/O connections so that the user can integrate sensors, actuators, communications, and MEMS chips to prototype the system. Although these prototypes will not be used for production releases, they play an important role in demonstrating a proof-of-concept and evaluating overall feasibility. Prototypes may have limited modeling fidelity to the final product characteristics, yet even rough estimates of size, performance, power/energy, and reliability can be important demonstrations of expected product feasibility.

In our driving application, the initial pre-prototype quickly identifies a need for greater performance than a CPU-only embedded system can provide, and thus we identify the need for a GPU- or FPGA-based system in order to meet performance goals. For the purposes of this article we defer the decision between GPU and FPGA- systems in order to demonstrate the design automation demands of both cases, but in a typical case the user would select either the GPU or FPGA-based system depending on the cost, performance and power constraints of the target system.

4.1. Device Driver Generation

IoT prototypes integrate a variety of additional chips for application-specific computations (e.g. AES encryption), sensors, actuators, RF communications, and MEMS devices. Although the prototyping platform is designed to make it easy to physically connect these chips, the user is still responsible for developing a software infrastructure and set of device drivers to integrate the chips into the IoT solution. Designing and integrating device drivers can be a major challenge even though chips use standard communications channels. Thus, automatic generation of platform device drivers for the software can significantly accelerate the prototyping process, allowing designers to concentrate on implementing software features. Manually written drivers often intermingle interactions with the OS and the device and thus complicate coding, maintenance, and require additional testing in the development cycle. However, specifications to interface with a device are OS-independent and can be auto-generated based on the manufacturer device and programming specifications. Furthermore, automatically generated drivers can also use generated test patterns to facilitate easier testing and verification of drivers.

4.2. FPGA-based Prototypes

IoT prototypes may use FPGA-based platforms for two similar and yet distinct purposes: a designer may target an FPGA with the intention of deploying an FPGA-based platform in the final product, or they may target an FPGA as a prototype of an expected full-custom SoC implementation of the application. Prototyping of an SoC allows full, real-time functional verification and timing verification. The FPGA-based prototype is critical for producing an inexpensive early validation of the platform design, but design automation is necessary for modeling, component selection, design space exploration, design entry and verification. Furthermore, prototyping of SoCs must consider that the achieved performance, power and size may not be as efficient as the final SoC. Automation in all of the design processes is important to make prototyping fast and efficient, and to simplify the process of translating the prototype into a design of the final SoC. Because the design automation needs are similar for both FPGA-based prototypes and custom SoCs, we will discuss these needs in further detail in Section 6.

5. IoT Design with Commercial Off-the-Shelf Components

IoT production device development with commercial off-the-shelf (COTS) components follows a similar process to prototyping, but with extra complexity in component selection, and more demands on modeling and evaluation of design objectives. A production device will optimize the physical size with packaging and a custom printed circuit board (PCB), but optimization for performance, power/energy, or other device features is based largely on component selection. The main custom feature of COTS-based devices is the PCB, and component selection, modeling and evaluation of potential system designs are key limiting factors in quickly and effectively designing platforms that meet device objectives.

5.1. High-level Modeling and Component Selection

Although a prior prototype may have served as a proof-of-concept, modeling of potential system level designs is important to evaluate whether a chosen components meet the device objectives. In particular, transaction-level modeling (TLM) in SystemC has become a popular approach for system modeling with high simulation speed. However, SystemC TLM models are often unavailable for components; when such models exist, it is important that they provide accurate power and performance estimates to maintain high fidelity between model estimates and achieved performance. Design automation can create SystemC TLM models from C-level specifications [31]; power and performance estimates may be based on manufacturer data, but may also require automation to generate estimates.

IoT systems may require modeling of not just digital components, but also the analog sensors, actuators, RF and MEMS components. These components not only require functional SystemC models but also detailed compatibility analysis. Whereas high-level modeling typically abstracts communication interface details, these components may require more detailed analysis to determine whether the components can be integrated. Analog sensor chips may have digital interfaces or require integration with Analog/Digital convertors before interfacing with a CPU; similarly other components may require verification of interface compatibility.

High-level modeling for IoT systems will be used to explore a variety of alternative system designs, with differing components and features. To reflect performance and power of the system, it is important to integrate with automated mapping of application software to the hardware platform; a system with a dedicated encryption chip may offload significant workload from the CPU, yielding either improved performance, more opportunity to turn off the CPU, or both. Design automation can track which resource(s) are suitable for each portion of the application and explore mapping decisions to determine the optimal mapping for a particular system. Automating this mapping is critical to allowing extensive design space exploration, as manual mapping would render the exploration too costly.

When high level modeling and automated software mapping are paired with design space exploration using a library of potential components, design automation can facilitate exploration of potential systems together with generation of a pareto-optimal set of designs with different combinations of design objectives. From this set of pareto-optimal designs, the user can more easily select a system that balances performance, power/energy, and cost of components. The selected design would include both a system-level design and an optimized mapping between application source and the system components.

High-level modeling and design space exploration plays a vital role in complex applications such as face recognition to identify the performance gap with CPU execution and selection of appropriate hardware accelerators. High-level models can also assist in software hardware partitioning decisions and also modeling and testing the interface between the CPU and the accelerator.

5.2. PCB Design

Although high level modeling typically abstracts communication details, C-level and SystemC models can be used to automatically generate detailed information on chip interconnect. When paired with chip specifications and automated PCB layout tools, design automation can be used to quickly generate initial PCB designs that can be refined and optimized. The complexity of PCBs in area, density, power dissipation, and total nets routed has been steadily increasing, placing pressure on design automation to assist in design and verification of board layouts.

6. Custom Platforms

IoT device development based on commercial off-the-shelf chips quickly reaches limits in performance, power/energy, and device size. In order to design more efficient IoT devices with tightly integrated chips, smaller printed circuit boards (PCBs), and lower overall cost per-device, producers turn towards custom platforms.

Custom Platform can either be System-in-Package (SiP) solution or a System-on-Chip (SoC) device. System-in-Package may refer to a variety of packaging technologies that tightly integrate multiple chip dies into a single package. The tight integration of multiple dies reduces power and energy of the devices, reduces the PCB size by integrating multiple chips into a single package, and can improve performance by reducing intercommunication latency. System-in-package designs may also include designs where the IoT device designer creates full-custom dies as part of the system design, it will then be an SiP that contains a custom SoC. However, we focus only the custom SoC in this section.

COTS and SiP-based systems allow comparatively fast development, primarily concentrating on software design and component selection when designing the IoT device. To achieve performance and power/energy efficiency infeasible with standard or existing platforms, developers turn to custom System-on-Chip solutions. In addition to improved features, performance and power, SoC-based solutions have the advantage of lower per-unit costs at volume.

System-on-chip devices have a variety of levels of customization, from lightly customized processors or IP-based design that integrates previously verified components to full custom designs that design entirely new CPU extensions or custom compute hardware. Although the level of customization does have an impact on the complexity of the design, and in turn the design automation needs, system-on-chip design in general increases required CAD complexity compared to demands of COTS-based systems. We generally classify SoC-based designs into two groups: CPU customizations that extend the instruction set of a CPU, and full custom designs that create standalone application specific hardware – sometimes with a CPU to handle control, error processing or interfacing. We will now talk about the design automation demands for these two strategies in detail.

6.1. CPU Customization

CPU customization retains compatibility with a prior instruction set but adds additional instructions to improve the performance and power efficiency of particular computations. For example, CPUs now commonly contain media or cryptographic extensions to make those styles of data-parallel processing more efficient. Custom CPUs not only improve performance and power efficiency, but also create feature differentiation and IP protection: a competitor cannot simply copy platform software because the ISA extensions require the custom CPU implementation.

6.1.1. Workload Analysis: In COTS-based systems, automated mapping of software to hardware platforms was needed to effectively perform system-level modeling and determine optimal performance and power/energy mapping the application to candidate system designs. However, here the workload analysis is orders of magnitude more complex; instead of mapping software at the granularity of large functions, we may develop instruction set extensions at the granularity of only a few instructions. Furthermore, to effectively use an ISA extension, we may require transformation of the application code for better loop organization, memory access patterns, or communications and data locality.

The process of CPU customization identifies not only a single ISA extension, but must select and evaluate multiple extensions considering both independent and joint benefit of a set of extensions as well as the benefits and costs of the extensions, which we will discuss in further detail in the following subsections. The enormous design space for CPU customizations makes it infeasible to evaluate more than a small subset of possible extensions, which places emphasis on effective design automation to analyse application source, identify potential extensions, and estimate benefit. Design automation in compilation techniques can find common repeated computation patterns to identify candidate extensions; when paired with polyhedral models that can transform loops, memory access patterns, and inter-iteration dependencies, this automation can play a key role in estimating the potential impact of an instruction set extension.

6.1.2. Modeling: Workload analysis is important to determine candidate extensions with maximal impact on the application source, but modeling of the performance and power/energy benefits of an extension is critical for decision making. An extension with high application coverage but little opportunity for performance improvement must be discarded. Conversely, even extensions with high potential for speedup must be evaluated relative to other extensions (or sets of extensions) that have lower cost in chip area or additional power. However, it is not feasible to perform detailed implementation of every candidate extension in order to perform decision making. CAD to automatically translate high level descriptions of extensions and generate area, performance, and power estimates are critical; these estimates must be fast and inexpensive to produce, yet have sufficient correlation with real implementation results to accurately guide decision making.

Automation in modeling must integrate synthesis of hardware for the extensions to generate area, performance and power estimates, evaluate what percentage of that area or power is design overhead (e.g. an extension that modifies the ALU should only consider area overhead, not total area), use automated mapping and compilation to estimate usage patterns, and estimate efficacy of power- and clock-gating on unused portions of the CPU (which may reduce total power instead of increasing). This represents the integration of multiple individually complex automation tasks, yet a necessary requirement for effectively determining which subset of ISA extensions represents the optimal tradeoff of area, performance, and power/energy for the IoT application.

In our driving face recognition application, the initial pre-prototype would have identified a need for greater performance than a CPU-only embedded system. Though ISA extensions can work well for certain compute-intensive but comparatively simple applications such as security (AES encryption), workload analysis and modeling of face recognition will point towards necessity of hardware acceleration with GPU or FPGAs.

6.1.3. HW Implementation: Workload analysis together with modeling determines a chosen set of CPU customizations; however, during implementation, it is critical that automation can assist in implementing the low level details so that area, performance, and power estimates can be achieved or improved on. High-level synthesis [32, 33] and automated IP- and system-integration [34] can fill an important role in performing detailed implementation. Many integration details are complex yet tedious and error prone. Automated integration can both improve design time and reduce verification effort as we will discuss next.

6.1.4. Verification and Debug: Verification and debug of customized CPUs can retain the verification/test vectors of the original design as an initial set. However, the CPU customizations not only create a new set of instructions that must also be verified in the hardware implementation, but also many potential corner cases depending on the complexity of modifications to any communications interfaces between existing and new (or modified) functional units. Furthermore, in IoT systems, integration with analog, RF, and MEMS components is a major component of verification. Instead of a simple set of instruction sequences, verification and debug must also explore possible cases of communication with these peripherals including A/D timing, interrupt handling corner cases, and verification of correct behavior under faulty external input.

Detecting, localizing and fixing any potential errors in this behavior can be extremely challenging. Detailed analysis of program execution with intermediate checksums can help discover implementation bugs and corner cases [35, 36]. However, these techniques can require exhaustive creation and comparison of checksums in large application code. Automation is thus critical to make the technique of detecting and localizing implementation bugs feasible.

6.2. Full Custom System-on-Chip IoTs

A custom SoC implementation delivering a powerful single silicon solution for an IoT device offers the best performance with significantly better energy efficiency than other platforms due to smaller form factors and lower power consumption. In addition, IoT product differentiation is possible by developing custom hardware for the proprietary features of the manufacturer, improved security and privacy by including accelerators for full-fledged encryption standards. However, one of the key challenges in development of IoT SoC devices is the long and iterative design cycle and is a bottleneck to meet stringent time-to-market requirements. The extensive design cycle also directly translates to higher NRE costs for design and development. Yet, custom SoCs present the best opportunity for designs with lowest per-unit cost at higher volume despite the challenges in the design flow. Thus, design automation of the SoC IoT flow is a critical need to reduce NRE costs and time-to-market in order to reduce risk and improve break-even point for IoT device volume.

We briefly introduced the challenges in developing a custom SoC IoT in our prior work [37]. Here, we expand and discuss in detail the design process and automation opportunities in the SoC IoT flow. The SoC design flow includes aggregating all requirements to create a specification, performing a high-level modeling to explore design space at module- and system-level, followed by a long and cumbersome process of implementation and an even longer verification and debug process.

6.2.1. SoC Modeling and Design Space Exploration: IoTs integrate multiple components including various analog sensors, actuators, digital and MEMS technologies, privacy and security modules, and communication components. It is important to perform system level evaluation of the proposed IoT device using high-level models and explore the design space to choose the design option that meets platform goals.

Design automation to model the entire SoC as a virtual platform using high-level languages is already a reality. In particular, transaction-level modeling (TLM) together with SystemC language has become a popular approach for SoC modeling, such high-level modeling improves simulation speed compared to RTL simulation and provides functional verification as well as early system modeling and analysis. A typical SoC modeling flow takes the system specification as input, which is often produced in C or C++ by software engineers. Then usually a manual hardware/software partitioning process is carried out, and the hardware portions are reimplemented using SystemC to work together with microprocessor IPs that target the software portion of the specification. Fast high-level accelerator modeling with accurate power and performance information is one critical building block in SoC modeling. The challenge is to obtain accurate power/performance information at early design stages without detailed implementation details. Accurate power information is usually not available until after logic synthesis or even physical design, which is too late for system-level modeling and analysis. The hardware design space is too vast to be explored thoroughly. Additionally, high-level models are typically written to achieve fast simulation speed, and not all of the parts are efficient or even feasible for high-level synthesis. To this end, an automated SystemC 3-stage modeling and synthesis framework [31] generates a high-level SystemC model annotated with power and latency estimations for accurate high-level performance and power modeling and another synthesizable SystemC model to enable HLS solutions. The framework also generates an analytical model providing power and latency information for all points in the design space and finally performs a fast design space exploration to generate pareto curves to guide effective low-power design. Custom SoCs with an embedded CPU create additional complexity with HW/SW codesign and partitioning and significantly increase the possible design space.

HW/SW Partitioning: Automation of HW/SW codesign is a pressing need for an efficient IoT design process. Although existing frameworks help to automate some of the profiling, design space exploration, and hardware characterization processes, IoT devices present additional challenges. First, IoT devices have extreme low power requirements; most devices will require and use clock-gating, power-gating, and DVFS as low-level mechanisms. Next, IoT devices have reliability, privacy and security requirements. These requirements may not be explicitly part of the high-level language specification; such specifications may be qualitative in nature or highly dependent on computation platform, thus requiring substantial effort to translate between software and hardware. These challenges significantly complicate the modeling and estimation of performance, power and area on both CPU and custom platforms, which can affect HW/SW codesign decisions. Thus, it will be important to develop fast and accurate estimation models that can incorporate both quantitative goals for area, performance, and power with reliability, privacy and security constraints.

The SoC modeling and hardware/software partitioning can accurately determine the application component that needs acceleration in our driving face recognition application. The components of face detection and creation of templates are most suited for hardware acceleration. The SystemC models will also help explore and identify the best interface for communication between the CPU and accelerator.

6.2.2. Circuit Implementation: Design automation for SoC implementation phase is possible by using HLS techniques that enable automated translation of high-level language descriptions such as C/C++, SystemC and CUDA to RTL [32, 33, 14] and/or by automated integration of several IP blocks including register transfer level (RTL) blocks.

High-Level Synthesis: The automatic translation to RTL through HLS substantially reduces design effort and expand design space exploration [38, 31], allowing fast and easy design of custom compute units. IoT devices commonly require small but efficient computation units to implement processing and analysis of data inputs from sensors. HLS is important not only to design such custom compute quickly, but also to allow designers to iteratively optimize algorithms and implementations quickly.

HLS has previously explored low-power design for control-flow intensive and data-dominated circuits, and activity reduction [39]. However, HLS for ultra-low power IoT designs requires automated application of clock-gating, power gating and DVFS technologies. These optimizations must also be balanced with performance and area in order to meet overall design constraints.

Privacy and security are critical for IoT devices to ensure that sensitive data is kept private and that IoT devices are secure from malicious remote control of such devices. HLS offers the ability to automatically integrate encryption IPs that secure input and output data streams, analyze input and output interfaces to ensure that every interface is secured, and allow the user to use software-tools to analyse the security of the system.

IP and System-level Integration: Custom hardware for IoT devices must be integrated into a system with sensors, actuators, CPU cores, and communications IPs. Through the use of standardized interfaces and protocols, custom hardware core integration can be fully automated so that a system-level design is produced through automated connection of IPs and custom components, substantially reducing manual effort to produce system level designs, control state machines, communication protocols, and testing infrastructure.

Furthermore, within the HLS produced core, a user may wish to use pre-defined, well optimized RTL IPs for important sub-functions. The use of these IPs accelerates the design process and ensures that the designer can meet system-level design goals in power, performance, area, privacy, and security; however, IP integration can be complex, time-consuming and error-prone as a manual process. Thus, HLS requires automation to effectively integrate IPs both within HLS-generated cores as well as through standardized interfaces at the system level. Prior HLS tools typically limit IP integration to a small set of provider defined IP cores; the user cannot specify custom IPs (either HLS-generated or RTL) to be integrated during HLS. Although HLS-produced cores often have standardized top-level interfaces, system-level integration is also left as a manual process. As a result, HLS-produced cores must be instantiated and connected with other system components manually, and designers must design appropriate control and glue logic to create the system level implementation.

Design automation should automate the process of instantiating, connecting and creating control and glue logic so that system level designs are quickly produced. Eliminating manual system integration can substantially improve design productivity and enable rapid system-level design space exploration.

As a solution to address the need of integration method in IoT design tools, IP integration within the HLS-generated core is proposed in [34], which, by directly specifying the IPs for implementing functions/instructions in high-level language specifications, effectively automates the processes involved in IP integration. IP integration within HLS-produced cores can substantially improve the design process. Instead of partitioning code so that IPs can be integrated at the system level, the HLS core directly integrates the IPs internally.

Furthermore, because system-level interfaces are commonly standardized, the HLS tool can assist in instantiating, connecting system level IPs and creating appropriate control state machines and glue logic between the system-level cores. Automation of IP integration within HLS cores and at the system level substantially improves the design process, and is a critical need for effective design of IoT devices.

In our driving face recognition application, if there exists a predesigned and verified face detection IP, it can be reused so that design effort can be focused on nodal analysis and template creation component.

6.2.3. Verification and debug: Although the initial design process is critical in the design flow, debug and verification time can be even more critical to time-to-market. The fraction of verification time as a percentage of the design flow has surpassed design time [40]. Verification effort is often a significant, labor intensive process. When a design is functionally incorrect, the engineer must manually identify the erroneous signal and trace backwards through simulations to find the source of the functional bug. Although HLS helps accelerate design time, the produced RTL code is not intended to be human readable or manually edited further exacerbating a manual verification process.

For these reasons it is critical to automate portions of the verification process in order to assist engineers in more quickly identifying functional errors. Automated instrumentation of HLS produced RTL is an active area of research, and helps users to gather trace data from executions on prototyping platforms. Although this assistance helps, these approaches still leave the problem of selecting which signals to trace to the user. Thus, although automated to help gather data, the more challenging tasks of selecting signals and identifying which signals are the source of functional error remains.

We develop a method [41] that automatically instruments applications, generates traces for every relevant operation type and inserts appropriate verification code into the output RTL such that simulations will automatically identify functional errors, pinpointed to the erroneous instruction, timestamp, and exact difference in expected value. This automation significantly aids engineers in quickly identifying the simulation source of functional error, which can be used to identify bugs in input source code more rapidly.

Furthermore, aside from functional debug, assistance in performance debugging is also critical. As an example, if a design is not meeting the throughput target (say II is not 1, or a FIFO is frequently full), it is usually challenging to pinpoint the underlying reasons in the source code. It is important for the HLS tool to localize the function, set of statements or coding styles that hinders meeting the performance constraints and in addition, assist the user with guidelines for restructuring the source code.

Verification and debug of a complex system such as face recognition is feasible only through design automation and effective CAD tools will enable faster design and creation of such premium and complex IoT systems.

7. Conclusions

In this article, we have highlighted the trends, demands and critical steps of the IoT design flow that requires CAD support for design of IoT devices. We described the development process of IoT devices in general, and then we examined current critical design automation needs for COTS-based, and custom platform devices. We also discussed both current missing needs for design automation as well as our ongoing work targeting these needs. Overall, CAD promises to reduce design time, cost and risk to serve as the bridge that makes design and implementation of custom platforms for IoT devices effective and timely.

8. References

[1] "Intel Edison," http://www.intel.com/content/www/us/en/do-it-yourself/edison.html.

[2] "Intel Galileo," http://www.intel.com/content/www/us/en/embedded\\/products/galileo/galileo-overview.html.

[3] "Microsoft .NET Gadgeteer," www.netmf.com/gadgeteer/.

[4] "Texas Instruments Internet of Things Featured Products," http://www.ti.com/ww/en/internet_of_things/iot-products.html.

[5] "Qualcomm Internet of Things Development Platform," https://developer.qualcomm.com/hardware/iot-cellular-dev.

[6] "NVIDIA TEGRA X1 Processors," http://www.nvidia.com/object/tegra-x1-processor.html.

[7] "NVIDIA TEGRA K1 Processors," http://www.nvidia.com/object/tegra-k1-processor.html.

[8] "Mali Graphics Hardware," www.arm.com/products/multimedia/mali-graphics-hardware/index.php.

[9] "AMD Embedded GPUs," http://www.amd.com/Documents/AMD_Embedded_Radeon_E8860_ProductBrief.pdf.

[10] *Nema Embedded GPU*, Think Silicon, http://www.think-silicon.com/product_Nema_GPGPU.php.

[11] "Altera User Customizable ARM-based SoC," https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/br/br-soc-fpga.pdf.

[12] "Xilinx ZYNQ All Programmable SoC," http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[13] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of fpgas and gpus for high productivity computing," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 94–101.

[14] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, W.-M. W. Hwu *et al.*, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *SASP*, 2009.

[15] S. T. Gurumani, H. Cholakkal, Y. Liang, K. Rupnow, and D. Chen, "High-level synthesis of multiple dependent CUDA kernels on FPGA," in *ASP-DAC*, 2013, pp. 305–312.

[16] H. Zheng, S. Gurumani, L. Yang, D. Chen, and K. Rupnow, "High-level synthesis with behavioral level multi-cycle path analysis," in *FPL*, 2013.

[17] *Vivado HLS*, Xilinx Inc, www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[18] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh, "From opencl to high-performance hardware on fpgas," in *FPL*, 2012, pp. 531–534.

[19] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *FPGA*, 2011, pp. 33–36.

[20] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "Autopilot: A platform-based esl synthesis system," in *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Netherlands, 2008, pp. 99–112.

[21] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[22] *NVIDIA Jetson TK1 Embedded Development Kit*, http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html.

[23] *Vivante Graphics Cores Product Brief*, www.vivantecorp.com/Product_Brief.pdf.

[24] "The Parallela Boards," https://www.parallella.org/board/.

[25] R. Krishnamurthy, "High-performance Energy-efficient Reconfigurable Accelerators/Co-processors for Tera-scale Multi-core Microprocessors," in *ARC*, 2010.

[26] *ABI research*, https://www.abiresearch.com/press/driven-by-increased-demands-on-healthcare-supplier/.

[27] M. T. Satria, S. Gurumani, W. Zheng, K. P. Tee, A. Koh, P. Yu, K. Rupnow, and D. Chen, "Real-time system-level implementation of a telepresence robot using an embedded gpu platform," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 1445–1448.

[28] "NEC Face Recognition," https://www.necam.com/biometrics/doc.cfm?t=facerecognition.

[29] "EUROTECH Face Recognition Systems," http://www.eurotech.com/en/products/devices/face+recognition+systems.

[30] "Ayonix Face Matcher," http://ayonix.com/products/ayonix-facematcher/.

[31] W. Zuo, W. Kemmerer, J. Bin Lim, L. Pouchet, A. Ayupov, T. Kim, K. Han, and D. Chen, "A Polyhedral-based SystemC Modeling and Generation Framework for Effective Low-power Design Space Exploration," in *ICCAD*, 2015.

[32] H. Zheng, S. Gurumani, L. Yang, D. Chen, and K. Rupnow, "High-level synthesis with behavioral level multi-cycle path analysis," in *FPL*, 2013.

[33] H. Zheng, S. T. Gurumani, L. Yang, D. Chen, and K. Rupnow, "High-level synthesis with behavioral-level multicycle path analysis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 33, no. 12, pp. 1832–1845, Dec 2014.

[34] L. Yang, S. Gurumani, D. Chen, and K. Rupnow, "Behavioral-Level IP Integration in High-Level Synthesis," in *FPT*, 2015.

[35] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra, "Qed: Quick error detection tests for effective post-silicon validation," in *Test Conference (ITC), 2010 IEEE International*. IEEE, 2010.

[36] K. A. Campbell, D. Lin, S. Mitra, and D. Chen, "Hybrid quick error detection (h-qed): Accelerator validation and debug using high-level synthesis principles," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015.

[37] L. Yang, Y. Chen, W. Zuo, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen, "System-Level Design Solutions: Enabling the IoT Explosion," in *ASICON*. IEEE, 2015.

[38] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *FPT*, 2011, pp. 1–8.

[39] Z. Zhang, D. Chen, S. Dai, and K. Campbell, "High-Level Synthesis for Low-Power Design," *IPSJ Transactions on System LSI Design Methodology*, vol. 8, pp. 12–25, 2015.

[40] H. D. Foster, "Trends in functional verification: a 2014 industry study," in *DAC*, 2015.

[41] L. Yang, M. Ikram, S. Gurumani, D. Chen, S. Fahmy, and K. Rupnow, "JIT Trace-based Verification for High-Level Synthesis," in *FPT*, 2015.