System Architecture Directions for Post-SoC/32-bit Networked Sensors

Hyung-Sin Kim, Michael P Andersen, Kaifei Chen, Sam Kumar, William J. Zhao, Kevin Ma, and David E. Culler
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
(hs.kim,m.andersen,kaifei,samkumar,william_zhao,kevinma.sd,culler)@berkeley.edu

ABSTRACT

The emergence of low-power 32-bit Systems-on-Chip (SoCs), which integrate a 32-bit MCU, radio, and flash, presents an opportunity to re-examine design points and trade-offs at all levels of the system architecture of networked sensors. To this end, we develop a post-SoC/32-bit design point called *Hamilton*, showing that using integrated components enables a ∼$7 core and shifts hardware modularity to design time. We study the interaction between hardware and embedded operating systems, identifying that (1) post-SoC motes provide lower idle current (5.9 μA) than traditional 16-bit motes, (2) 32-bit MCUs are a major energy consumer (e.g., tick increases idle current >50 times), comparable to radios, and (3) thread-based concurrency is viable, requiring only 8.3 μs of context switch time. We design a system architecture, based on a tickless multithreading operating system, with cooperative/adaptive clocking, advanced sensor abstraction, and preemptive packet processing. Its efficient MCU control improves concurrency with ∼30% less energy consumption. Together, these developments set the system architecture for networked sensors in a new direction.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Sensor networks*; System on a chip;

KEYWORDS

Wireless Sensor Network, System-on-Chip, Mote, System Architecture, Operating System, Multithreading, RIOT, OpenThread

ACM Reference Format:
Hyung-Sin Kim, Michael P Andersen, Kaifei Chen, Sam Kumar, William J. Zhao, Kevin Ma, and David E. Culler. 2018. System Architecture Directions for Post-SoC/32-bit Networked Sensors. In *The 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18), November 4–7, 2018, Shenzhen, China.* ACM, Shenzhen, China, 14 pages. https://doi.org/10.1145/3274783.3274839

1 INTRODUCTION

We consider the central question of whether and how system architecture issues for embedded wireless sensor network (WSN) systems in the years ahead may be fundamentally different from those that have guided design over the past 20 years, since the emergence of the field [31, 37]. While it is easy to observe high-level trends and infer that at some point conventional concurrency models and networking stacks will fit on this class of computing platform, when dealing with low power embedded operation, *the devil is always in the details.*

The datasheets for modern 32-bit System-on-Chip units (SoCs) with an integrated microcontroller (MCU) and radio are truly impressive [10, 43]. Thread-based OSes have reemerged (e.g., RIOT [14]), and commercially-driven IPv6 network protocols have arrived (e.g., OpenThread [71]). However, cursory analysis would suggest that they still fail to obtain the extremely low idle-power, fast wake-up and context switching, efficient active operation, and low cost that are intrinsic requirements of long-lived, unattended WSNs operating on a limited energy budget [37]. This has been achieved by communication-centric, event-driven execution on relatively primitive MCU, radio, ADC, and transducer hardware [64].

We present a detailed design, implementation and analysis to better answer this question. This involves: (1) creating a new hardware platform, Hamilton [9], focused on design for manufacturability (DFM), (2) porting both an event-driven OS (Contiki [25]) and a thread-based OS (RIOT) to the hardware platform, (3) carefully optimizing and re-implementing a thread-based OS to attend to power and concurrency issues at the level previously obtained with event-driven execution, and (4) conducting whole-system analysis of the resulting power and performance characteristics. With all of these efforts, the higher level question can be answered:

- Modular platforms with application-specific sensor subsystems separated from an expert-designed processing-radio-storage core are replaced by *design level modularity* of a few highly integrated components dropped as a design block onto a cheap, rudimentary PCB along with digital sensors.
- With meticulous, platform-aware design of the 'idle thread', idle current on 32-bit SoC-based (e.g., Atmel SAMR21 [10]) platforms can, after 15 years, be brought well below that achieved by TinyOS [64] on 16-bit TI MSP430-based platforms (e.g., TelosB[75]).
- With high clock-rate and advanced memory system, context switching among threads, on either yield or preemption, is now faster than event handling on 16-bit platforms. Advanced hardware support for peripherals vastly diminishes the concurrency demands on the system. Preemption becomes critical as processing-intensive analytics are incorporated locally into applications.
- With idle listening and other techniques incorporated into radio hardware, assumptions that radio power consumption vastly exceeds that of the CPU and that the CPU can busy-wait are no longer valid. Instead, system design should focus on offloading

SenSys '18, November 4–7, 2018, Shenzhen, China

ACM ISBN 978-1-4503-5952-8/18/11…$15.00
https://doi.org/10.1145/3274783.3274839

operations to intelligent peripherals and coordinating these operations while sleeping in between. The incorporation of ADCs and processing directly into digital sensors extends this trend.

To illustrate design issues that are likely to arise in this new system architecture regime, we explore several questions. With the availability of multiple SoC clocks and clock/power domains, how can these be coordinated to achieve the best idle, wake-up, and active power characteristics? With the prevalence of intelligent, multi-mode digital sensors, how should the basic primitives and abstractions evolve? In the presence of preemption, relatively limited buffering, and latency sensitivity, how should concurrency in the embedded network stack be structured?

To frame the investigation, Section 2 provides a brief retrospective of critical design factors in generations of sensor system platforms, i.e., motes. Section 3 presents a specific, open-source post-SoC design point that reduces the manufacturing costs below $7 and essentially eliminates (previously dominant) assembly costs. It is represented as a design block that can be dropped into application-specific systems. Section 4 uses this platform to investigate embedded OS issues, showing why a tickless scheduler is necessary to achieve a true idle mode (with a current draw of 2.6 μA for the core and 5.9 μA for a full sensor suite), why the MCU control becomes as important as the radio control for low power operation, and why the thread concurrency model becomes competitive with an event-driven approach. Together these establish the transition to a new sensor system design regime. Section 5 presents a post-SoC architecture with careful MCU management and efficient preemptive multithreading: cooperative/adaptive clocking, sensor abstraction, and preemptive packet processing. Section 6 provides an overall evaluation using this platform, optimized RIOT, and a redesigned OpenThread stack under typical WSN scenarios, demonstrating concurrency improvements with ~30% less energy consumption. Section 7 discusses remaining issues and Section 8 concludes.

2 RELATED WORK: CO-EVOLUTION OF MOTE AND WSN TECHNOLOGY

In a traditional system architecture [37], there are five distinct characteristics of networked sensors: (1) Small physical size and low power consumption, (2) Concurrency-intensive operation, (3) Limited physical parallelism and controller hierarchy, (4) Diversity in design and usage, and (5) Robust operation. Constraint (1) shaped the evolution of *motes* for two decades [91] and motivated WSN system designs, which are clearly differentiated from that of embedded devices, such as Raspberry Pi and Arduino.

8-bit Motes (the First Generation). By 2000, a state-of-the-art mote [37] had an 8-bit MCU with a 4 MHz clock (calibrated by an external crystal) and 0.5 kB RAM. This resource-constrained MCU needed to handle every peripheral-generated event due to lack of independent controllers. For example, its radio had no buffer, forcing the MCU to handle modulation of *each bit* to/from the radio *on time*. This significant overhead on the MCU motivated a concurrency-intensive OS, TinyOS [64]. To provide high concurrency with slow processing and small memory, TinyOS adopted an event-based concurrency model with a single shared stack, a tickless scheduler, and non-preemptive run-to-completion semantics. This made the event model (rather than the thread model) *standard practice* for WSNs.

8-bit motes evolved further. Mica [38] reduced the MCU burden by handling radio data in *8-bit chunks* and automating transmission timing. A peripheral bus approach (featuring a 51-pin connector) provided modularity. MicaZ [68] additionally reduced the burden on the MCU by (1) handling radio data in *128-byte packets* and (2) further automating communication operations. Its TI CC2420 radio [33] supported the IEEE 802.15.4 standard, had two 128-byte buffers and handled all physical and some MAC layer operations autonomously. The CC2420 became a representative radio in WSN and spawned numerous low-power network protocols [18, 24, 69].

16-bit Motes (the Second Generation). In 2004, the TelosB [75] began the 16-bit mote era and significantly impacted system architecture with the following features: (1) More RAM (10 kB RAM in the TI MSP430 [40]) enabled storage and processing of sensor data in larger batches [78], reducing concurrency requirements. (2) Integration of every component on a *single board* with careful power gating achieved a low-power, easy-to-use, and robust mote, while sacrificing modularity. The combination of the MSP430 and the CC2420 became a common choice for 16-bit motes, such as the Z1 [94], Epic [30], and kMote [49]. While Z1 is a complete board including sensors like the TelosB, the kMote and Epic motes separate the sensor board from the core for modularity.

In 2008, Dutta *et al.* observed three stages (Prototype, Pilot and Production) of the WSN lifecycle. While the prototype and pilot favor modular hardware, production favors design-time hardware inlining [30]. This conclusion was affirmed by several platforms [8, 38, 49, 87, 89], following the 'reusable core' and 'application-specific daughter board' architecture.

With 16-bit motes, a number of studies suggested a compromise OS architecture: *a thread model for applications and an event model for the kernel*. Contiki [25] provided a tick-based event kernel with *preemptive* multithreading as a library. Protothreads [26] provided *cooperative* multithreading to relieve programmers of the burden of synchronization. Protothreads were *stackless* to save memory, but this prevented using thread-local variables or blocking inside of a routine [67]. To alleviate this problem, TinyThread [67] introduced *stack-based* cooperative multithreading, with a static stack analysis tool [77] to mitigate stack overflow.

As computationally intensive algorithms, such as data compression [54, 78], were applied in the WSN setting, cooperative multithreading became problematic: a long-running computation in an application can adversely affect system concurrency [56]. To alleviate this issue, TOSThread [56] proposed *stack-based preemptive* multithreading for applications, on top of the event-driven kernel of TinyOS. The authors observed that context switching overhead is lower in networked sensors than in general-purpose computers because they do not support virtual memory.

32-bit Motes (the Third Generation). Several early efforts sought to achieve a SoC or 32-bit mote, including Jennic [46], Imote, and Imote2 [21], but failed to achieve low power operation. In 2012, Egs and Opal showed that a 32-bit mote can save both time and power compared to 16-bit motes when performing complex computations, but idle current remained higher [57]. In 2016, WandStem [87] showed a 32-bit core achieving lower idle current than 16-bit motes. Firestorm [8] showed that a 32-bit mote can be applied to the maker

space by providing abundant extension pins like the Arduino and both IEEE 802.15.4 and BLE radios.

RIOT [14] returned to a classical stack-based preemptive multithreading concept *in both the kernel and userspace* [93] for 32-bit motes in 2013, with message-passing IPC (interprocess communication) and a tickless scheduler. The latest 32-bit MCUs have memory protection units (MPUs). Recent work [8, 65] investigated the potential of MPUs, but used a hybrid concurrency model: processes for the application and events for the kernel.

Emergence of SoCs. Today, the sophisticated core components such as the MCU and radio are integrated into a single unit, a SoC. Although the emergence of SoC comes from system integration factors, not the MCU progression per se, in WSN the SoCs including a 32-bit MCU dominate. For example, the OpenMote [89] is built on the TI CC2538 [41], an early 32-bit SoC with high energy consumption by both the MCU and radio. Its latest revision, the OpenMote B [72], takes a single-board approach like TelosB, but uses the same SoC. The Atmel SAMR21 [10] and TI CC2650 [43] are more recent low-power SoCs, which provide much lower active current for the MCU and radio than the CC2538. The CC2650's radio has its own 32-bit MCU, further extending the controller hierarchy.

Several boards have been built on SAMR21, none of which are low power. SAMR21-XPRO [11] is an evaluation board used in several works [48, 74, 83], which consumes high idle current due to its many peripherals. Sparrow [22] also consumes high idle current (30 µA). CC2650STK [42] is a complete sensor board built on the CC2650, which is low power but provides a limited development environment (only board flashing and packet reading) [39]. However, *a SoC-based mote design has not yet been studied systematically.*

Overall, two decades of technical advancement have dramatically impacted two characteristics of WSNs: (1) concurrency-intensive operation and (2) physical parallelism and controller hierarchy. This has motivated new WSN system designs. Likewise, the emergence of SoCs presents new opportunities. To realize their potential, a post-SoC system architecture needs to be formulated and evaluated.

3 HAMILTON: A 32-BIT, POST-SOC MOTE

To ground our study, we develop Hamilton (Figure 1) [9], an open source, 32-bit SoC-based mote. Off-the-shelf SoC-based boards, such as OpenMote [89] and SAMR21-XPRO [11], do not permit systematic study of design characteristics in a post-SoC era, because they do not offer application-specific design for manufacturing (DFM), low power operation, or a sensor suite appropriate for analysis.

3.1 Core

The core encompasses everything that traditionally defines a mote: processing, storage and communication. In Hamilton, the core is built around a SAMR21 SoC that combines a 32-bit Cortex M0+, an AT86RF233 802.15.4 radio, and (optionally) serial flash all in a single package for $4.58. This SoC provides ample resources (32 kB RAM/256 kB flash) which allows the use of relatively heavy protocol stacks, such as OpenThread [71]. Alternatively, the CC2650 has less memory (20 kB/128 kB) but adds BLE for the same price.[1] Both have comparable power characteristics listed in their datasheets.

[1]Adding a BLE module is out of scope of this paper but we provide a brief discussion in Section 7.

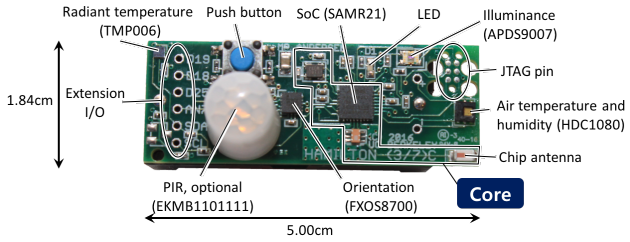


Figure 1: Hamilton, a single board that has a low-power 32-bit SoC and provides 7 types of sensors.

	Storm	Hamilton
MCU	7.06 USD	4.58 USD
Flash	3.18 USD	
Radio	2.8 USD	
RF frontend	2.73 USD	0.91 USD
Oscillators	0.57 USD	0.66 USD
Assembly	12.00 USD	0.6 USD
Total	30.00 USD	6.75 USD

Table 1: Cost for manufacturing the cores of Storm [7] and Hamilton. SoC significantly lowers assembly cost.

The SoC is combined with an off-the-shelf *chip antenna* and a *fully-integrated balun* specifically designed for the AT86RF233. Unlike many previous motes, such as TelosB, Epic, and OpenMote, we do not use an external oscillator to feed the MCU clock, which reduces cost. This is because internal oscillators in recent SoCs are fairly accurate and can be calibrated by the crystal for the radio [8]. Hamilton uses a 3.0V CR123A Lithium battery as its power source.[2] Compared to two 1.5V AA alkaline batteries, a higher-energy-density Lithium primary cell provides more consistent voltage for a longer time (important for transducers that do not operate at lower voltages) and reduces the device size.

The BOM (bill of materials) costs of a SoC-based design are predictably lower (e.g., the SoC costs half as much as its constituent MCU and radio bought independently) than those without a SoC. However, the greatest benefit in post-SoC design is that the costs of *PCB manufacture and assembly*, which have typically driven the overall cost of a mote, are significantly reduced, as described below.

Previously, a dense circuit with compact (e.g., ball grid array) packages required thin traces, multiple PCB layers and small vias, all of which increased manufacturing costs. In contrast, a SoC-based mote has fewer chips, no traces for interconnect between the MCU, radio and flash, and fewer passive components. Previous motes used expert-designed discrete-element circuits for the balun and antenna matching network [30], which required precision-value passives and tight process control in manufacturing; This approach is no longer required as fully integrated, cheap baluns are available off the shelf. Similarly, a PCB trace antenna has long been the standard approach to avoid costly SMA connectors and bulky whip antennas, but this requires expert design and expensive impedance controlled PCB manufacture, along with extensive design validation across multiple prototype manufacture runs. Now, off-the-shelf chip antennas perform well, even with no impedance matching network.

[2]Although this battery is used for valid reasons, our Hamilton design is independent from any specific choice of battery.

Overall, the Hamilton core consists of *only five off-the-shelf components* with five low-precision capacitors: SoC, balun, chip antenna, crystal (for the radio), and battery. With significantly lower circuit complexity,[3] it can be manufactured without advanced techniques, for *less than $7*, even at small quantities. Table 1 compares the total cost of Hamilton to that of the Storm [7], a recent 32-bit mote without SoC, showing that Hamilton is 77% cheaper. More importantly, while assembly is the most significant part of manufacturing Storm's cost (40%), it becomes the least significant in Hamilton (8.8%).[4] Using a SoC greatly reduces assembly overhead and makes BOM costs the dominant factor in overall cost.

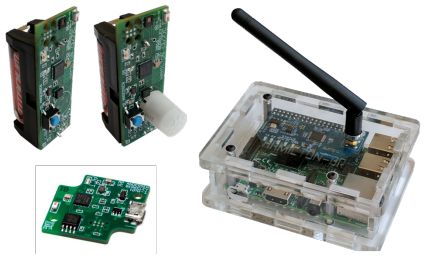
3.2 Design for Manufacturing (DFM)

Recognizing the significantly reduced design complexity and manufacturing cost of the post-SoC mote core, we must revisit the question of *modularity* to support a wide diversity of designs. Specifically, the tolerances of a SoC-era core board with its reduced complexity are comparable to the tolerances demanded by digital transducers. These tolerances are well within the capacity of maker-friendly small-run PCB fabrication facilities, such as Seeed Studio [4] which can produce 20 four-layer Hamilton PCBs for $2 each with no tooling and setup costs.[5] It is now cheaper, simpler and faster to produce a fully integrated mote than to produce a 'sensor daughterboard + mote' combination even at small quantities.

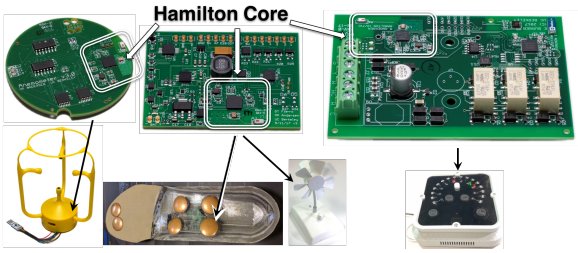
Modularity is still key, but reuse occurs at the *design level*. Free CAD software packages such as Autodesk Eagle [12] now support *design blocks* allowing a core to be dropped into an application-specific design incorporating sensors. In addition to the reduced complexity of the core, design-time application specialization removes all the traces and vias associated with unused I/O pins, further reducing complexity and manufacturing cost. This is in contrast with the expansion ports on conventional modular designs which must accommodate the maximum conceivable I/O demands [8, 38, 49].

Modern digital sensors are now affordable, presenting additional opportunities. Rather than analog conditioning circuitry for each transducer (e.g., potential dividers and filters), digital transducers integrate everything, including the ADC, into a low-pinout package. Furthermore, they connect to a bus, reducing the number of traces on the board and required MCU pins. Also, in the past, complex techniques such as multiple power domains at different voltages or individual component power gating, were required to achieve whole-board low power operation. In contrast, modern digital sensors provide advanced low-power modes, and power consumption can be adequately controlled using digital commands.

To quantify the impact of DFM in a post-SoC mote, we drive the *specialization* of the Hamilton core, following the 'fully integrated single board' architecture like the Z1 and TelosB, with typical environmental monitoring as an application. According to feedback from users, such as civil engineers and building architects, we include transducers to sense the following: air temperature, humidity, radiant temperature, illuminance, magnetic field, acceleration, and optionally PIR motion. We include JTAG via bed-of-nails connector as the user interface, which is fast and provides rich debugging features with no additional BOM cost or power consumption. This platform is the basis for the OS and networking investigations in the subsequent sections.



(a) WSN ecosystem: 1) with 6 sensors, 2) with 7 sensors (+PIR sensor), 3) small powered router, and 4) border router with a raspberry pi



(b) Extended ecosystem: 1) ultrasonic anemometer, 2) auto-heating/cooling insole and smart desk fan (both with an actuator), and 3) thermostat

Figure 2: Hamilton ecosystem. The low manufacturing burden in the post-SoC era shifts hardware modularity to the design time, enabling easy production of various boards using the same core.

Manufactured in the USA at prototype quantities (50), each complete system costs $25 for turnkey production (PCB manufacture, assembly, programming and calibration), about $8 more than the BOM cost. Manufactured in China at pilot quantities (500), the complete system costs $18, only $2 more than the BOM cost. At production quantities (10k), each complete system is quoted at $12, which is below the BOM cost if purchased at that quantity in the USA. With low manufacturing cost and the additional advantages, such as a smaller form factor, increased reliability, and improved power efficiency [75], it is reasonable to follow the single-board approach instead of the 'core + daughter board' approach or off-the-shelf development kits.

Figure 2 illustrates Hamilton's hardware ecosystem. The core has been used not only in the typical WSN setting (Figure 2(a)) but also in several projects including an anemometer, wearable device, desk fan, and thermostat (Figure 2(b)) [16]. Hardware modularity at the design time allows a single-board architecture for all devices.

3.3 Summary

This section investigated design and manufacturing aspects when using modern components, including overall cost. The Hamilton core and its environment sensing specialization are open source design blocks[6] — anyone can drop them into an application schematic

[3]For comparison, the TelosB core requires an MCU, radio, flash, two crystals, and a carefully designed balun and PCB antenna, with 28 passive components [70].
[4]We used the same manufacturing vendor for fair comparison.
[5]In contrast, without SoC, e.g., Storm [8], a discrete-component PCB requires 6 layers and tight tolerances, resulting in each PCB cost an order of magnitude more with a minimum quantity of 1000 and hundreds of dollars in setup and tooling costs.

[6]Hamilton design blocks are provided at https://github.com/hamilton-mote/hw and Hamilton motes are commercially available at https://hamiltoniot.com.

	Idle Current
TelosB/TinyOS	8.9 µA
Storm/TinyOS	13.0 µA
Firestorm/TinyOS	25.6 µA
TelosB/Contiki	36 µA
OpenMote/Contiki	2169 µA
CC2650STK/Contiki	13.2 µA
Hamilton/Contiki	347 µA
TelosB/RIOT	1926 µA
SAMR21-XPRO/RIOT	6012 µA
Hamilton/RIOT	5.9 µA (2.6 µA core)

Table 2: Average idle current consumption of various device/OS packages with 3.3V of input voltage (5.0V for SAMR21-XPRO).

and send the files to a turnkey manufacturer. The latest technology has crossed a critical threshold regarding assembly burden, opening a low-cost 32-bit mote era. Without advanced techniques, post-SoC motes can be manufactured by more accessible PCB vendors. Low-cost motes have the potential to facilitate dense deployment and enable researchers with a low budget (common in developing countries) to more easily enter into this regime.

4 CHARACTERIZING POST-SOC HARDWARE/OS INTERACTION

In this section, we establish critical performance characteristics of Hamilton (power consumption and concurrency) when interacting with embedded OSes, and perform a systematic comparison to those of prior motes. To this end, we port both RIOT [14] and Contiki[7] on Hamilton and use popular mote/OS packages, such as TelosB/TinyOS, TelosB/Contiki, and OpenMote/Contiki, as benchmarks. We use the latest Contiki-NG [5] for the Contiki evaluation.

4.1 Power Consumption: Idle Current

Given that a networked sensor is expected to sleep most of the time, idle current dominates battery lifetime. Although idle current of each hardware component is provided by its datasheet, idle current of a mote/OS ensemble must be measured directly since it may be impacted by many additional factors, including board assembly and OS behavior, making it significantly greater than the sum of the datasheets values. We measure the average idle current of various mote/OS packages, as shown in Table 2.

TelosB/TinyOS consumes 8.9 µA of idle current, consistent with previous studies [55]. Storm consumes slightly more current than TelosB, showing that a 32-bit mote was less energy efficient than TelosB a few years ago. Firestorm uses Storm as its core but adds more components, such as BLE, and uses a larger board, doubling the idle current.

Interestingly, the idle current of TelosB/Contiki is 4 times higher than that of TelosB/TinyOS, 36 µA. We found that, unlike TinyOS, Contiki's *tick-based* scheduler makes the MCU periodically wake up to poll and process if any pending task exists (e.g., timer expiry and I/O request). The tick period must be short (around a few ms) for responsive operation, increasing power consumption. The other two cases of Contiki, OpenMote and CC2650STK, also exhibit an idle current that is much higher than what is given in the SoC datasheets [41, 43]. This confirms that the OS's MCU control impacts power consumption of networked sensors.

Idle current of OpenMote/Contiki (or CC2650STK/Contiki) is much higher (or slightly lower) than that of the TelosB/Contiki. This variation among boards is partly because each driver implementation has a different tick period, but also comes from the hardware's power characteristics. While OpenMote's CC2538 SoC requires an MCU active current 6.5 times higher than MSP430 [89], the CC2650 has slightly higher MCU-active current. This shows that 32-bit SoC power characteristics have significantly improved.

RIOT has a tickless scheduler and supports low power operation by using a dedicated *idle thread*. The idle thread is executed whenever the MCU is free, transitioning the MCU to the lowest power mode permitted by the current peripheral state. However, the idle thread does not work as intended, at least in the case of TelosB and SAMR21 drivers. We observed an idle current of 1.926 mA for TelosB and 19.7 mA for Hamilton. Achieving a low idle current is not just about novel concepts but careful driver implementation considering every component's behavior.

We undertook the effort to implement a true idle thread in RIOT, fixing bugs in the MCU, radio, and SPI drivers. Finally, we observe that, with a low power 32 kHz clock running in the idle mode, Hamilton/RIOT consumes 5.9 µA, which is 34% lower than the TelosB/TinyOS case. With only the core of Hamilton (no sensors), the idle current is 2.6 µA. Although we did not include complex power-gating circuitry for assembly, the idle current due to sensors is only 3.3 µA. This confirms that, in the Post-SoC era, the low power modes integrated in the latest digital sensors are sufficient for low power operation of an entire board; there is no need for expensive manufacturing to obtain a low-power mote.

Even with our improved SAMR21 driver, the SAMR21-XPRO/RIOT case shows a very high idle current (6.012 mA) due to more peripherals running by default. A low-power SoC does not necessarily result in a low-power board: The entire board design should be low-power focused. Finally, Hamilton/Contiki consumes 347 µA with the same tick period as the TelosB driver, two orders of magnitude greater than the Hamilton/RIOT case. We confirmed that Hamilton/Contiki provides the same idle current (5.9 µA) when disabling the tick (but this renders Contiki inoperable). It does show that a *true* low-power idle mode for networked sensors should completely eliminate the active operation of the MCU.

4.2 Power Consumption: MCU vs. Radio

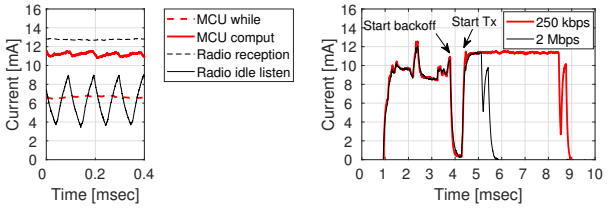
The most significant energy consumer on a mote has historically been its radio, motivating numerous low power networking protocols [18, 24, 69]. It was a reasonable trade-off to reduce the radio's duty-cycle by increasing the MCU's duty-cycle (more computation or busy-waiting). For example, MAC implementations on TinyOS (BoX-MAC-2 [69]) and Contiki (ContikiMAC [24] and TSCH [28]) make the MCU *busy-wait* during the radio idle listening for timely radio control (i.e., turning it off as soon as possible).

This tradeoff must be revisited in the Post-SoC era because the 32-bit MCU power draw is fairly high, while the radio power draw has been reduced. Table 3 shows the power ratio of the radio to the MCU for various motes [8, 42, 75, 89]. Most 16-bit motes use

[7] The Contiki/Hamilton port is not fully connected to the network stack but functional enough for this study.

	Idle listening/ MCU active	Tx (0 dBm)/ MCU active
16-bit motes (4 MHz)	9.40	8.70
OpenMote (32 MHz)	1.82	1.85
CC2650STK (48 MHz)	2.21	2.28
Storm (48 MHz)	1.37	1.37
Hamilton (48 MHz)	0.98	1.82

Table 3: Ratio of current consumption on radio operations to that on MCU's active mode. Modern 32-bit motes have much lower ratio than traditional 16-bit motes.



(a) Smart idle listening (b) High data rate (2 Mbps)

Figure 3: Current waveform of various Hamilton operations, demonstrating the impact of Hamilton's advanced radio features on power consumption.

the (early version of) MSP430 MCU and the CC2420 radio, which have a 9.4 power ratio for idle listening. However, 32-bit motes provide significantly lower power ratios, due to lower radio power and higher MCU power.

Specifically, the Hamilton radio provides advanced features, such as smart idle listening and high data rate transmission (up to 2 Mbps). Figure 3(a) shows that with smart idle listening, the radio current waveform becomes saw-toothed, implying that it tries to sleep as much as possible.[8] This reduces idle listening current to only half of receiving current, as opposed to previous motes where idle listening and receiving consume similar current. The average idle listening current (6.38 mA) becomes slightly lower than the MCU active current (6.50 mA, when busy-waiting by executing the 'while' statement), resulting in the power ratio of 0.98. It also shows that the current for MCU computation (i.e., AES encryption in this case) is now similar to that of radio reception.

Figure 3(b) compares normal (250 kbps) and high (2 Mbps) data rates when Hamilton wakes up, broadcasts a packet, and sleeps. Interestingly, the high data rate reduces transmission time *without additional current*. It significantly reduces radio energy in exchange for reduced transmission range (~7 dBm loss). This figure also reveals automatic CSMA/CA, where the radio sleeps during the backoff to save energy.

Overall, the design and evaluation of a low-power network protocol for post-SoC motes must jointly consider MCU and radio power consumption. In the Hamilton case, making the MCU busy-wait during idle listening *doubles* power consumption. Likewise, a low-power OS should remove any redundant MCU wake-up.

4.3 Concurrency: Context Switch

Next, we investigate concurrency: events vs. threads. **The event model** supports highly concurrent operations and is resource efficient [37] but requires finite state machine-based programming and manual matching of *calls* and *returns* [6], increasing the programming burden [86, 90]. **The thread model** enables a more natural programming style, with the blocking-wait abstraction and easy pairing of calls/returns [90], but requires context switches that are heavier than event dequeuing and has greater memory overhead in the form of per-thread stacks [37].

[8]The exact radio operation with the smart idle listening is not open to the public.

	Preempt	Yield
TelosB/TinyOS (4 MHz)	38.9 µs	21.0 µs
TelosB/Contiki (4 MHz)	87.6 µs	72.0 µs
Hamilton/Contiki (48 MHz)	5.84 µs	4.36 µs
TelosB/RIOT (4 MHz)	92.2 µs	83.0 µs
Hamilton/RIOT (4 MHz)	79.4 µs	74.6 µs
Hamilton/RIOT (48 MHz)	8.30 µs	7.76 µs

Table 4: Context switch times. *Preempt* refers to waking up a thread or posting a task, and then switching to that thread/task. *Yield* refers to switching to a thread/task that is already runnable/posted.

Although the thread model is widely used in general-purpose computers, familiar to most system programmers (taught in most, if not all, undergraduate OS classes) and extensively investigated in academia and industry, networked sensors have used the event model (e.g., TinyOS and Contiki) due to significant resource constraints of the underlying hardware. Some limited forms of threading have been proposed, such as Protothread for Contiki [26] and TOSThread for TinyOS [56], which provide user-level thread-based abstractions rather than full stack-based preemptive thread model. However, it has been two decades since TinyOS was developed and a decade since the above hybrid models were developed. Meanwhile, motes have become more capable. It is therefore meaningful to investigate whether the thread model is viable in this regime.

To this end, we measure the time to switch tasks (or threads) in TinyOS/ Contiki (or RIOT). Table 4 shows that, with TelosB, TinyOS provides the fastest task switch. Contiki is more than twice as slow as TinyOS, due to more background processing, but is still faster than RIOT. An event-driven model has lower overhead than a thread-based model.

Advanced hardware, however, can afford the context switch overhead of the thread model. Hamilton/RIOT provides a shorter context switch than TelosB/RIOT with the same clock speed (4 MHz) due to the efficiency of a 32-bit MCU and its memory system. With the 48 MHz clock, Hamilton/RIOT's context switch delay becomes much shorter, ~9 µs. This is 3 times shorter than TelosB/TinyOS and only ~3 µs longer than Hamilton/Contiki. Given that the task switch time of TelosB/TinyOS has been regarded as reasonable, the modern MCU is fast enough to compensate for the inefficiencies of threads, making stack-based multithreading viable for concurrency. This likely applies to not only 32-bit and/or post-SoC motes but any mote which has a capable-enough MCU. For example, modern 16-bit MCUs, such as MSP430FR5994 [44], also have fast clock and large memory space.

4.4 Concurrency: Computation Offloading

The use of DMA and modern peripherals allows computation to be offloaded from the MCU, which significantly relieves concurrency requirements. Hamilton's radio provides automatic CSMA/CA and link layer retransmission and automatically sleeps during CSMA

	No DMA	DMA
ADC Sample Time	643 µs (busy)	599 µs (idle)
I2C Read Time	587 µs (busy)	492 µs (idle)

Table 5: ADC and I2C operations on Hamilton/RIOT. With DMA, data can be read from peripherals more efficiently, without consuming CPU time.

backoff for energy savings.[9] This enables the MCU to be idle during the entire frame transmission without managing backoff and radio mode.[10] For comparison, the popular CC2420 radio requires the MCU to wake up many times during a frame transmission: the number of CSMA tries multiplied by the number of link-layer transmissions.

In addition, given that digital sensors (five sensors in Hamilton) have their own ADCs, the MCU only needs to read registers to obtain prepared results. By using DMA, the MCU can be idle when reading these values, or even when reading from an analog sensor through the ADC. We write a DMA controller driver on RIOT and measure the transfer of a 2-byte reading from the illumination sensor through the ADC and a 4-byte reading from the humidity sensor via the I2C bus. Table 5 shows that the DMA relieves the MCU from computation for a nontrivial amount of time.

4.5 Summary

Our preliminary study of a post-SoC mote identifies the following *paradigm shifts* in the system architecture for networked sensors:

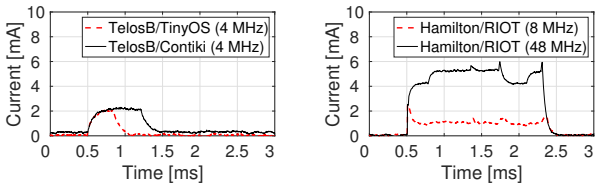
- A post-SoC mote, despite having more resources, supporting more advanced features, and being more easily manufacturable, provides lower idle current than (carefully designed and manufactured) traditional motes.
- MCU power consumption has become comparable to radio power consumption, making MCU power management important. A low-power OS must be tickless and remove any redundant MCU operation (e.g., busy-waiting). A low-power network protocol should consider both the MCU and the radio power consumption.
- The full thread-based concurrency model is now viable in this regime because (1) a modern MCU can afford to context switch overhead, and (2) automated modern peripherals relieve concurrency requirement.

5 POST-SOC SYSTEM ARCHITECTURE

Motivated by the given paradigm shifts in the previous section, we investigate a post-SoC system architecture with deep consideration of modern hardware characteristics. Specifically, we aim to establish design principles by answering the following questions:

- How can we utilize a modern MCU in an energy efficient way without sacrificing its fast processing capability?
- How can we provide efficient interaction between the MCU and an automated modern peripheral and what is the benefit?
- What benefits arise from using the full thread-based concurrency model, beyond familiarity?

[9]The radio cannot receive packets during this auto-CSMA/CA procedure since it sleeps during the backoff. This feature is recommended for leaf nodes rather than routers.
[10] Although not included in Hamilton, off-the-shelf BLE radios are even more independent of the MCU, providing automatic time synchronization, channel hopping, and duty-cycling [60, 85].



(a) TelosB/TinyOS and TelosB/Contiki (b) Hamilton/RIOT (default)

Figure 4: Waveform of simple wake-up and sleep operation.

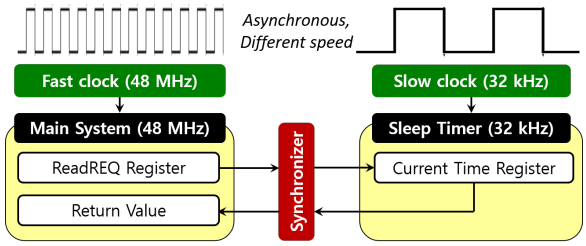


Figure 5: An example of interaction between different asynchronous clock domains (the case of SAMR21).

5.1 Cooperative Clocking

We first focus on modern MCUs having various different clock domains and investigate the challenges and opportunities. To this end, we measure the latency of the MCU wake-up and sleep transitions, which significantly impacts the duration of the MCU active period. We program a simple application which periodically wakes up and immediately sleeps again. Two clock speeds are tested for Hamilton: 8 MHz and 48 MHz. The 8 MHz clock consumes less active power than the 48 MHz clock, depicted in Figure 4.

Figure 4(a) shows that TelosB/TinyOS and TelosB/Contiki take 0.35 ms and 0.70 ms, respectively, from waking up to sleeping again (except for the power draw tail). Contiki does more background work at each wakeup, resulting in a longer active period than TinyOS. Interestingly, Figure 4(b) shows that although Hamilton has faster system clocks than TelosB, the Hamilton/RIOT cases require active periods of 1.81 ms and 1.88 ms (except for the 0.23 ms long power draw tail), much longer than the TelosB cases. Given that it takes only 0.02 ms to wake up the hardware, most of the time is for the software execution. Further, the 48 MHz clock does not significantly reduce the active period compared to the 8 MHz clock. Given that context switch delay is negligibly short in Hamilton, this active period should be shortened.

Investigating further, we observe that timer management consumes 87% of the active period. Most of the time is consumed getting the current time value, which is necessary for setting a new timer, updating an existing timer, and confirming a timer expiry. Given that the low-power 32 kHz clock runs during the idle mode, the timer management procedure for wakeup and sleep requires the current time of the *low power clock* rather than the fast main clock. Since the main clock and the low power clock are asynchronous and have significantly different speeds, accessing the low power timer register requires crossing clock domains [88]. On the SAMR21, the synchronizer circuit performs this as depicted in Figure 5. It introduces a long access delay and lengthens the active period despite the fast main clock.

To address the challenges posed by heterogeneous clock domains, we design a *cooperative clocking* mechanism. Given that both the main and low power clocks are operating in the active period and the main clock information is fast to access (synchronous clock domain), we use the two clocks together in a *cooperative* manner. Specifically, we access both clock domains once after each wakeup to get a reference time for each clock. Then, we find the time of the low power clock *indirectly* by using the main clock and the reference information, as shown in Eq. (1) where t_{LP} and t_{main} represent the time of the low power and main clocks respectively, and f_{LP} and f_{main} represent the frequency of the two clocks.

$$t_{\text{LP}}(\text{now}) = (t_{\text{main}}(\text{now}) - t_{\text{main}}(\text{ref})) \cdot \frac{f_{\text{LP}}}{f_{\text{main}}} + t_{\text{LP}}(\text{ref}) \qquad (1)$$

As a result, interaction between the two clock domains happens *only twice* during the active period, right after waking up to synchronize the clock domains and right before sleeping to set a sleep timer. The reduced timer management overhead shortens the active period and allows the MCU to perform other tasks quickly (see Section 6.1).

5.2 Adaptive Clocking

Next, among the various clocks of a modern MCU, what clock should be used as the main system clock to maximize energy efficiency (e.g., 48 MHz clock vs. 8 MHz clock in the case Hamilton)? A "time vs. current" trade-off exists with the main clock: a faster clock reduces time but increases power consumption. It is known that a faster clock is more energy efficient in computationally intensive operations [8, 57], but we take a more comprehensive view.

The MCU is required to do two types of jobs: computation and interaction with peripherals. Although the computation period depends on the main clock, the interaction period does not; it depends most heavily on the method used for communicating with the peripheral, such as ADC, I2C, SPI, and synchronizer. For example, I2C speed (400 kHz) and different clock domain access (Figure 4(b)) are much slower than the main clock and determine the interaction period; using a faster main clock results only in higher current. On the other hand, encryption and SPI speed are limited by the main clock speed, so using a faster clock with DMA feature can save both time and energy.

Therefore, we use the fast clock *opportunistically*, when communicating with peripherals where the main clock speed matters (e.g., SPI in SAMR21), and when doing heavy computation, such as en/decryption and packet processing. Otherwise, the slow main clock is used to minimize active current. In this way, the modern MCU finds the sweet spot of the tradeoff between fast/high-power and slow/low-power clock, which improves both energy efficiency and processing delay.

5.3 Sensor Abstraction and Driver

We observed a gap between modern sensors and the sensor abstraction in low-power OSes. Specifically, although it is common for the application layer to request a sensor to read one type of sensor information at a time, modern sensors can often acquire multiple types of information at once.

To address this gap, we implement a simple caching mechanism for sensor drivers. When the application requests sensor information, the sensor driver has the sensor perform its natural suite of samples (e.g., both temperature and humidity information at once),

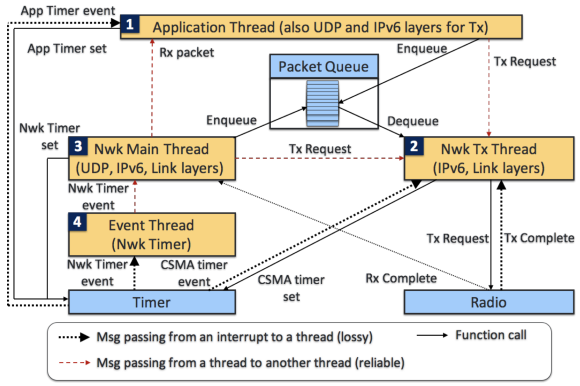


Figure 6: Proposed preemptive network architecture

but returns only the requested type of information to the application (e.g., only temperature); it caches the other information (e.g., humidity). If the requested type of information is freshly cached, the sensor driver simply returns the cached value without sensing. This reduces the number of sensor activations and register accesses, which saves energy and reduces concurrency requirements. Users can configure a time threshold to remove the outdated cached information.

Furthermore, some raw sensor values are hard to interpret. They need to be converted to meaningful units through nontrivial computation. However, given that end applications read sensor data from a server, instead of directly from a networked sensor device, the conversion process can be offloaded to the server, resulting in more energy savings.

5.4 Preemptive Network Architecture

Beyond its familiarity to programmers, stack-based preemptive multithreading in the *kernel* can benefit timely radio control and packet processing without sacrificing MCU power.

Timely radio control has always been important for low-power MACs to turn off the radio as fast as possible. Furthermore, given that the radio buffer can usually hold only one received packet [8, 75, 94], it is important to quickly move the packet to the MCU memory space before reception of the next packet begins. Otherwise, packet loss results even with good link quality and sufficient queue capacity. This problem becomes severe when a node needs to receive many packets while performing other tasks, such as packet transmission and computation [86]. For example, nodes near the border router should relay heavy traffic in a large-scale network [51, 53]. Active monitoring applications let each node store data and trigger *bursty* transmission by sending a query. Machine or structural health monitoring needs frequent data gathering, such as vibration and acceleration [47, 54].

To address this issue, we design a *preemptive network architecture*, as illustrated in Figure 6. Our architecture has four threads: network main (priority 3), network Tx (priority 2), application (priority 1), and event (priority 4) threads where a higher priority thread preempts a lower priority one. Each thread has its own message queue, and context switching is triggered by message passing between threads or from an interrupt handler to a thread. There is a packet queue shared by the three threads, which must be locked for synchronization.

- **The network main thread** incorporates UDP, IPv6, 6LoWPAN, and link layers and handles most network operation, such as information updates and processing received packets. It receives an `Rx_Complete` message from the radio upon a packet reception. Then it *preempts* the application and the network Tx threads to process the received packet, resulting in fast packet processing *right after the radio receives it*. When the received packet is for the application, it processes IPv6 and UDP layers and passes the payload to the application thread. When generating or relaying a packet, it adds the packet to the packet queue and sends a `Tx_Request` message to the network Tx thread.
- **The network Tx thread** incorporates IPv6, 6LoWPAN, and link layers and handles packet transmissions: from getting a packet from the packet queue to sending (including routing and CSMA/CA) and dequeueing it. This thread receives a `Tx_Complete` message from the radio at the end of each transmission, which includes transmission success, CSMA/CA failure, and transmission failure. It starts the whole procedure once receiving a `Tx_Request` message from the network main and/or the application threads and repeats until the packet queue is empty.
- **The application thread** handles application operation. When generating a packet to send, it processes UDP and IPv6 layers, enqueues the packet, and sends a `Tx_Request` message to the network Tx thread. It receives the application payload of a received packet from the network main thread.

Inter-layer function call. Although it is intuitive to provide one thread for each network layer, we include multiple network layers in one thread (e.g., UDP, IPv6, and 6LoWPAN in the network main thread). This is because function call is more efficient than thread context switch for inter-layer communication [20] and having fewer threads can save memory space.

Thread starvation. At a glance, this Rx-focused preemptive architecture would seem to risk the starvation of the network Tx and application threads when the network is fully loaded (nearly continuous packet reception). However, the radio operation is highly independent and we design the MCU to avoid busy-waiting. Therefore the MCU can perform other tasks (e.g., computation at the application thread) while the radio is transmitting or receiving a packet, improving concurrency. This independent radio operation, along with the modern MCU's fast processing capability, prevents the starvation of the other threads, shown in Section 6.2. Note that the fast modern MCU packet processing delay (up to 2.05 ms/0.65 ms with/without security processing) is shorter than the low-power radio receiving time (up to 4.66 ms with 250 kbps, including ACK transmission).

Message queue size. It is important to set each thread's message queue size considering both memory space and robustness. The network main and application threads pass a `Tx_Request` message to the network Tx thread only when there are no queued `Tx_Request` messages, which reduces the number of context switches. Given that `CSMA_Timer` (before transmission) and `Tx_Complete` (after transmission) events do not occur simultaneously, the message queue size of the network Tx thread is bounded by 2 (one for `Tx_Request` and the other for `CSMA_Timer` and `Tx_Complete`). As for the network main thread, given that it sets only one `Nwk Timer` (the most urgent one) at a time, it expects to receive only one `Nwk Timer` event. Given fast MCU processing speed, it does not expect to have many `Rx_Complete` messages. We bound the message queue size of the network main thread to 4, one for Nwk_Timer and the others for `Rx_Complete`.

Comparison with other OSes. When the radio receives a packet while the MCU is idle, Contiki adds a poll event and waits for the next system tick to extract the packet from the radio and process it, which couples delay for packet extraction/processing and radio control with the tick period.[11] Contiki's CSMA MAC suffers from this delay. To alleviate the problem, Contiki's low-power MACs (ContikiMAC and TSCH) make the MCU *busy-wait* while the radio is idle listening, which incurs nontrivial energy costs.

Without tick, TinyOS moves a packet from the radio to the MCU memory immediately after its reception (in an interrupt context) and posts a task to process the packet. However, because of run-to-completion semantics, the packet processing needs to wait until other pre-posted tasks are processed (i.e., fast extraction but slow processing).

RIOT's IPv6 stack, called GNRC, provides a thread for *each network layer* and allows a lower layer thread to preempt a higher layer thread. When the radio receives a packet, an interrupt handler simply informs the link layer thread of the event, making both packet extraction and processing happen in a thread context. Although the link layer thread preempts higher layer threads to quickly respond to radio events, a packet reception cannot preempt a packet transmission because both of them are executed within the link layer thread. The speed of extracting and processing a received packet depends on how many other packets are queued for transmission. Furthermore, RIOT's 'thread per layer' approach[12] requires 7 threads, which causes more context switches than in our architecture.

5.5 Thread-based Safe I/O Processing

Event-driven OSes have a large global task queue which is sized to avoid overflow. However, in the stack-based thread model, each thread has a local, relatively small message queue which can overflow. In RIOT, if a thread passes a message to another thread with a full message queue, the sending thread blocks and waits until the destination thread's queue becomes available (i.e., back pressure). However, when an interrupt handler has the same situation while sending a message to a thread, it must *drop* the message, which may cause severe system problems. For example, in the case of our architecture, if an interrupt handler passes a `Nwk_Timer` event directly to the network main thread, it can be dropped when the network main thread's message queue is full of `Rx_Complete` events.

To avoid missing events coming from an interrupt handler, we add another thread, called the event thread.[13] **The event thread**

[11] Note that general purpose tick-based OSes, such as the previous designs of FreeBSD [45] and Linux [84], use tick only for timer operation and execute threads asynchronously regardless of tick. In contrast, the use of ticks is ingrained in the Contiki/Protothread design [25, 26]. Any asynchronous interrupt can only *pend* corresponding protothreads, which are executed on the next tick, resulting in less responsiveness. Given that other tick-based OSes do not do this, we do not see a fundamental reason to use the *synchronous* protothread management, other than implementation simplicity.

[12] The 'thread per layer' design was criticized due to its inefficiency [20].

[13] Alternatively, we can give the network main thread multiple message queues [15], where one queue handles timer events and the other queue handles received packets.

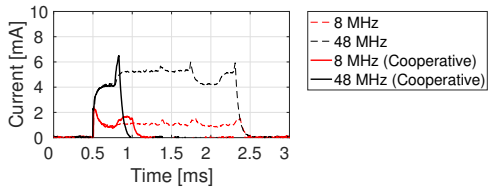


Figure 7: Waveform of simple wake-up and sleep operation with Hamilton/RIOT. It is highly impacted by utilization of various clocks in the 32-bit MCU.

manages `Nwk_Timer` events, except the CSMA timer. It receives an event message from an interrupt handler and delivers it to the network main thread. When the network main thread's queue is full, the event thread, acting as a bridge, blocks and waits until the network main thread's queue becomes available again, providing reliable message delivery. The event thread expects to receive only one timer event, which bounds its message queue size to 1. Due to its simple operation, the event thread's stack size can be small, like the idle thread.

5.6 Summary

We present design examples to show improved energy efficiency and concurrency in the post-SoC era. Cooperative and adaptive clocking can be applied to any MCU having multiple heterogeneous clocks, while their effectiveness will vary depending on MCU (clock architecture) and OS (timer implementation). The preemptive network architecture is hardware agnostic and can be applied to any full thread OS, but its effectiveness may also vary according to the radio.[14] The idea of robust I/O can be applied when a thread should process heterogeneous tasks and its message queue can be full of a specific type of task and lose an event from an interrupt handler. The sensor driver is a hardware-specific optimization, which shows the potential of modern peripherals. Overall, we see this work as a stepping stone for efficient system design in the post-SoC era.

6 EVALUATION

This section evaluates the post-SoC system architecture. We apply our architecture on RIOT and OpenThread [71].[15] The clocking mechanisms and sensor abstraction are implemented within RIOT drivers. The preemptive network architecture is implemented on RIOT/OpenThread.[16] Given that timer and radio are the resources shared by multiple threads (as in Figure 6), we modified RIOT to make timer access and radio mode change *atomic* (interrupt free).

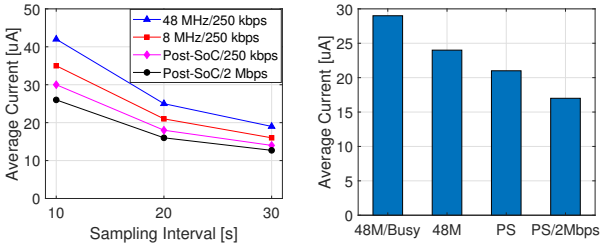
6.1 Energy Consumption

Figure 7 shows the results of cooperative clocking on Hamilton/RIOT in a simple wakeup and sleep scenario. The active period is significantly reduced: the 48 MHz and 8 MHz cases provide 0.33 ms and 0.50 ms of active periods respectively, which are 82% and 73% shorter than before. The gap in active periods between the 8 MHz and 48 MHz cases is slightly increased because some computation

[14]For example, the SAMR21 SoC's radio has a single-packet buffer for both Tx and Rx [10], the CC2420 radio has a single-packet buffer for each of Tx and Rx [33], and the CC2650 SoC's radio has a multi-packet buffer [43].

[15]OpenThread is an official open implementation of Thread [36], a multihop low-power network recently standardized from an industry consortium, which uses UDP/IPv6/6LoWPAN.

[16]Our code is open at https://github.com/hamilton-mote/hamilton-sw-sensys18.



(a) Periodic sensing and sending (b) Periodic receiving (every 10 sec)

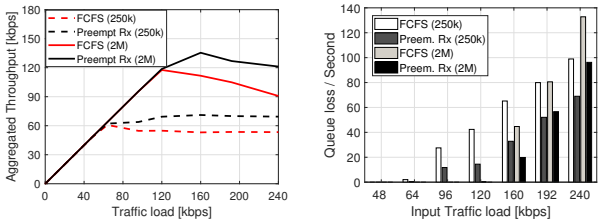
Figure 8: Average current consumption of Hamilton/RIOT/OpenThread with various settings.

is needed for clock updates. Given that the effectiveness of cooperative clocking can vary depending on hardware architecture and timer implementation, the dramatic performance improvement in Figure 7 partly comes from the RIOT's choice for its timer implementation: It checks the current time information more than of TinyOS and Contiki for accurate timer expiry. In addition, the active period with cooperative clocking is shorter than the TelosB/TinyOS case (Figure 4(a)), which shows that careful OS design can mitigate the pitfalls of complex clock domains in a post-SoC mote to save time and energy.

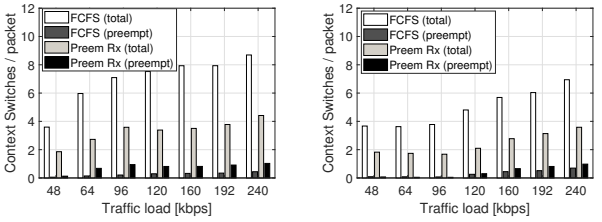
Next, we evaluate energy consumption in a sensing and sending application: a Hamilton periodically wakes up, reads the information from all six sensors, generates an AES-128 encrypted payload with the readings, broadcasts a UDP packet, and sleeps. Figure 8(a) shows the energy consumption of the four cases according to the sampling interval: 8 MHz and 48 MHz main clocks with default RIOT and 250 kbps data rate, and our post-SoC system design with 250 kbps and 2 Mbps data rate. The results of the three 250 kbps cases show that the post-SoC architecture reduces energy consumption by 14% and 29% for the 8 MHz and 48 MHz cases, respectively, increasing lifetime by 3 years. Given that the three cases consume the same energy on the radio, this verifies that the MCU energy savings (by advanced clocking and sensor abstraction) have a large impact on lifetime. Using a high data rate further reduces energy consumption by 13%. Lastly, note that the post-SoC architecture provides average current below 14 µA when the sampling interval is 30 seconds. Recalling the very high *idle* current of Contiki in Table 2 (e.g., 347 µA for Hamilton), this verifies again that a system tick should be avoided for 32-bit motes.

In addition, we evaluate energy consumption of a duty-cycling node when it wakes up, receives a packet from its parent router,[17] and sleeps every 10 seconds (Figure 8(b)). The '48M/Busy' case uses a 48 MHz main clock on default RIOT, and the MCU busy-waits during idle listening (mimicking Contiki's MAC implementations). 17.7 ms elapse between wakeup and completion of processing for the received packet and sleeping, resulting in a 29 µA average current draw. When the MCU actually idles during idle listening periods (the '48M' case), current draw is reduced by 17%, to 24 µA, which demonstrates the impact of decoupling the MCU operation from the radio on energy consumption. When our post-SoC architecture is applied (the 'PS' case), it takes 15.5 ms to finish a periodic task

[17]OpenThread uses a polling-based duty-cycling MAC where a leaf node periodically sends a data request to the parent router and listens to the channel if the parent has packets to send [36, 79].



(a) Throughput (b) Queue loss at the relay node



(c) Context switch overhead at the relay node (250 kbps) (d) Context switch overhead at the relay node (2 Mbps)

Figure 9: Performance in a three-node line topology where the border router receives packets from the other two nodes.

and the current consumption is further reduced to 21 µA, due to the improved clocking mechanisms. Lastly, when the radio uses a high data rate (the 'PS/2Mbps' case), each operation period is reduced to 11.4 ms and the current consumption decreases to 17 µA. Overall, with the same network protocol, using the post-SoC architecture and high data rate reduces energy consumption by 42%.

6.2 Concurrency and Packet Delivery

To evaluate the effect of preemptive reception, we configured a three-node line topology where the border router (Raspberry Pi and Hamilton connected through serial with 1 Mbps baudrate) receives packets periodically sent from nodes that are one and two hops away, respectively. As traffic load increases, the one-hop (middle) node becomes the bottleneck since it must receive/send the two-hop node's packets in addition to generating/sending its own packets. For comparison, we implemented a nonpreemptive reception where one network thread processes both Rx and Tx in a first-come, first-served (FCFS) manner.

Figure 9 shows various performance metrics according to total input traffic load (sum of the two nodes' traffic). Figure 9(a) shows that the preemptive reception achieves more aggregate throughput than the FCFS case. The performance improvement becomes larger with a 2 Mbps data rate, because both the border router and the relay node quickly process received packets in the radio buffer, allowing the radio to continuously receive packets. Otherwise, a packet sender will experience more CSMA backoff without receiving an ACK from the receiver, which causes more delay and queue loss as shown in Figure 9(b).

We confirm this by observing context switch overhead in Figures 9(c) and 9(d). FCFS causes many more context switches per delivered packet due to more CSMA backoffs. In contrast, preemptive reception provides significantly lower context switch overhead. Under heavy traffic, it always causes one *preemptive* context switch for relaying a packet, verifying that the MCU is busy doing other tasks whenever its radio buffer has a new received packet. This *one*

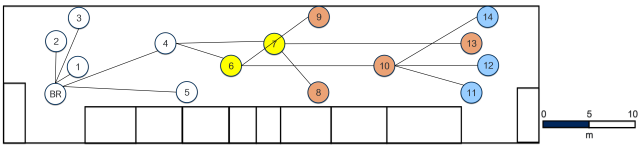
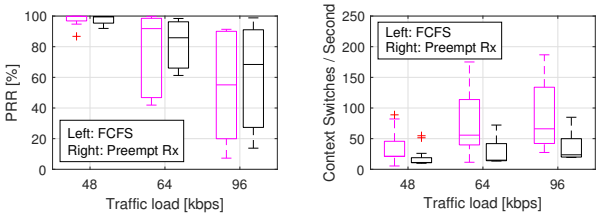


Figure 10: Testbed topology with a snapshot of 4-hop upward routing paths given by OpenThread when using transmission power of -6 dBm. Nodes with the same hop count have the same color.



(a) PRR of each node (b) Context switches of each node

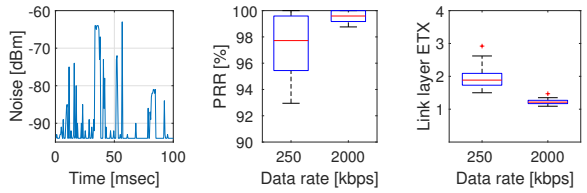
Figure 11: Performance on a testbed where each node periodically sends packets to the border router.

preemption nullifies redundant delay and significantly improves system efficiency.

In addition, we measure the MCU's duty-cycle and confirm that it is at most 50% even at the border router (the busiest node) when the throughput is saturated. This confirms that the main bottleneck of throughput performance is not the MCU processing speed but the radio communication and the radio-MCU interaction. Although the MCU is idle enough due to its fast processing speed and the radio's highly independent operation, preemption still makes a difference by improving concurrency and enabling the MCU to utilize the radio in a timely manner. This low duty-cycle also verifies that the MCU is capable enough to perform computation at the application thread while it is busy doing network operation.

For multihop networks, we configured a testbed in an office where one Hamilton border router and 14 Hamiltons form a 4-hop topology with OpenThread as in Figure 10. We first evaluate the performance when each node periodically sends packets to the border router with the same packet interval. Figures 11(a) and 11(b) plot packet reception ratio (PRR) and context switch overhead according to total input traffic load. The experiments were performed during the nighttime without duty-cycling to focus on the effect of preemptive reception. The two figures verify that the observations from Figure 9 are also valid in multihop networks. Preemptive reception delivers more traffic with fewer context switches. While preemptive packet reception leads to high throughput in this scenario, it is important to note that the primary value of preemptive reception is not just high throughput, but more efficient *concurrency* in various scenarios. For example, in other scenarios, better concurrency may result in less energy consumption.

We evaluate the 250 kbps and 2 Mbps data rates by having each node send a packet every 30 sec during normal working hours. Figure 12(a) shows that the 2.4 GHz band becomes very noisy at this time, even without WiFi (Bluetooth devices have become pervasive). Figures 12(b) and 12(c) show that using a high data rate causes significantly fewer transmission failures, resulting in significantly better PRR. A higher data rate makes each packet



(a) Noise measurement in channel 26, at node 1

(b) End-to-end PRR of each node

(c) Link layer ETX of each node

Figure 12: Performance when each node sends a packet to the border router every 30 sec. It shows that high data rate is effective for avoiding interference.

shorter, which reduces the likelihood for a packet to experience interference. Although the 2 Mbps date rate in Hamilton reduces transmission range by 7 dBm, it does not change the routing path in our environment. Recalling the results in Figure 8, a high data rate helps save energy and avoid wireless interference, which shows additional potential of post-SoC motes.

7 DISCUSSION: GOING FORWARD

This work began at the start of the post-SoC era. Now across a critical threshold, hardware technology continues to evolve rapidly. Low-power 32-bit SoCs are becoming more capable with clock rates up to 64 MHz, 256 kB RAM and 1 MB flash with low idle current [81], reminiscent of i386 in early PCs. They have separate processors and buffers for managing sensors and radios [43], further reducing the burden of the main CPU. Radios are becoming crystal free [92] and even more energy efficient (i.e., an order of magnitude reduction compared to CC2420 [50]), and support multiple antennas [10]. Their auto-security processing now supports both confidentiality and integrity (i.e., AES-CCM [28]). Some recent motes even have a public-key cryptography module [82]. It has become increasingly common for a SoC to offer both 802.15.4 and BLE radios. The latest Bluetooth 5.0 [76] supports 2 Mbps data rate without additional current consumption. *System design must evolve to match these technological advances*, rather than sticking to old customs, such as "extremely limited implementation to operate on TelosB".

For example, it has been common that an implementation of a low-power Internet protocol standard (e.g., 6LoWPAN, RPL, and embedded TCP) supports only a subset of the whole standard due to resource constraints, making it not interoperable with reasonable performance [13, 52, 58]. But given more memory space in post-SoC motes, the "interoperability vs. complexity" question needs to be revisited. Advanced radio features also present new research opportunities, such as adaptive data rate control (high vs. low rate) for energy efficiency and interference avoidance, and multi-antenna diversity [59] for reliable communication. Public-key crypto modules enable to explore asymmetric cryptography [82]. As both 802.15.4 and BLE support frequency hopping [27], mesh networks [35, 61], and open sourced low-power OS [85], comparison or joint use of BLE and 802.15.4 in a SoC needs investigation. For dependable evaluation of all the potential [17], migration of the reference hardware in open WSN testbeds needs to be discussed, given that some representative testbeds mainly use traditional motes [23, 80].

Open source operating systems present a landscape of shifting tradeoffs. TinyOS has robust and efficient system design [19, 34, 56, 63], but has a steep learning curve [62]. Its use has faded, with active maintenance ceasing about 6 years ago [1]. Contiki's system design is relatively inefficient but is widely used [17, 52] with implementations of more recent network protocols [24, 27, 29, 32], a COOJA simulator [73], and active maintenance [5]. RIOT has a full thread model [93] and supports more boards with more maintenance effort [3], but does not yet have an efficient and robust design (e.g., timer and message queue). Its network stack has not been thoroughly tested. Recently growing Tock [65] supports safe multi-programming with use of MPU, Rust language [66], and an event-driven kernel, but its network stack and open source community are yet at an early stage. There are some open-source multihop network stacks, such as OpenWSN [2] and OpenThread [71], which can run by themselves (event-driven kernel) or on embedded OSes. Once we understand the pros and cons of the various embedded OSes, can we create a useful synergy of their approaches?

8 CONCLUSION

As we move into the post-SoC era, several of the guiding principles of WSN research [37] have shifted: (1) With reduced manufacturing burden, hardware modularity is replaced by design-time modularity where application-specific motes are constructed using design blocks and fabricated using readily-available and affordable turnkey manufacturing. (2) As the MCU energy consumption is now significant on a mote, low power OSes and network protocols should consider the MCU more carefully. Tickless operation and avoiding busy-wait are now necessary. Cooperative/adaptive clocking significantly reduces energy consumption on the MCU. When applying all the above techniques, post-SoC motes become even more energy efficient than traditional motes with lower idle current, shorter active period, and lower energy consumption during the active period; The time of "low power vs. high performance" [57] has gone. (3) There is no longer a lack of resources and physical parallelism that dictates the concurrency model. It is now viable to return to the familiar thread-based model. Moreover, the thread model gives a chance to investigate a preemptive network architecture. The preemption enables efficient interaction between the radio and the MCU, improving both concurrency and energy efficiency.

We provide the Hamilton design block, the Hamilton itself, and our changes to RIOT and OpenThread. To revise an old adage, one might say "WSN is dead, long live WSN with a post-SoC system architecture."

ACKNOWLEDGMENTS

This research is supported in part by the Department of Energy Grant No. DE-EE0007685, California Energy Commission, Intel Corporation, in part by the National Science Foundation (NSF) Grant No. CPS-1239552, Fulbright Scholarship Program, and University of California, Berkeley, and in part by the Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education (NRF-2016R1A6A3A03007799). Additionally, this material is based upon work supported by the NSF Graduate Research Fellowship Program under Grant No. DGE-1752814.

We thank Ali Moin, Corten Singer, Bradley Cage, and Simon Duquennoy for their help on porting Hamilton on Contiki, RIOT maintainers for their useful feedback on RIOT, Albert Goto for helping build the Hamilton testbed, and John Kolb for his thorough review of this manuscript. We are thankful to the anonymous reviewers and the shepherd for their valuable feedback.

REFERENCES

[1] [n. d.]. Main development repository for TinyOS (an OS for embedded, wireless devices). ([n. d.]). https://github.com/tinyos/tinyos-main.

[2] [n. d.]. OpenWSN firmware. ([n. d.]). github.com/openwsn-berkeley/openwsn-fw.

[3] [n. d.]. RIOT - The friendly OS for IoT. ([n. d.]). https://github.com/RIOT-OS/RIOT.

[4] [n. d.]. Seeed Studio, the IoT Hardware Enabler. ([n. d.]). https://www.seeedstudio.com/.

[5] 2017. Contiki-NG, the Next Generation Contiki. (2017). https://www.contiki-ng.org//.

[6] Atul Adya, Jon Howell, Marvin Theimer, William J Bolosky, and John R Douceur. 2002. Cooperative Task Management Without Manual Stack Management.. In *USENIX Annual Technical Conference, General Track*. 289–302.

[7] Michael P Andersen and David E Culler. 2014. System Design Trade-offs in a Next-Generation Embedded Wireless Platform. In *Technical Report UCB/EECS-2014-162*. EECS Department, University of California, Berkeley.

[8] Michael P Andersen, Gabe Fierro, and David E Culler. 2016. System Design for a Synergistic, Low Power Mote/BLE Embedded Platform. In *Information Processing in Sensor Networks (IPSN), 2016 15th ACM/IEEE International Conference on*. IEEE, 1–12.

[9] Michael P Andersen, Hyung-Sin Kim, and David E Culler. 2017. Hamilton: a Cost-Effective, Low Power Networked Sensor for Indoor Environment Monitoring. In *Proceedings of the 4th ACM International Conference on Systems for Energy-Efficient Built Environments*. ACM, 36.

[10] Atmel. 2016. Atmel SAM R21E / SAM R21G, SMART ARM-Based Wireless Microcontroller. (2016). https://www.mouser.com/ds/2/268/Atmel-42223-SAM-R21_Datasheet-1065540.pdf.

[11] Atmel. 2016. SAM R21 Xplained Pro User Guide. (2016). https://http://www.atmel.com/Images/Atmel-42243-SAMR21-Xplained-Pro_User-Guide.pdf.

[12] Autodesk. [n. d.]. Eagle. ([n. d.]). https://www.autodesk.com/products/eagle/overview.

[13] Hudson Ayers, Paul Crews, Hubert Teo, Conor McAvity, Amit Levy, and Philip Levis. 2018. Design Considerations for Low Power Internet Protocols. *arXiv preprint arXiv:1806.10751* (2018).

[14] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. 2018. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal* (2018).

[15] Gaurav Banga, Jeffrey C Mogul, Peter Druschel, et al. 1999. A Scalable and Explicit Event Delivery Mechanism for UNIX.. In *USENIX Annual Technical Conference, General Track*. 253–265.

[16] Fred Bauman, Paul Raftery, Joyce Kim, Soazig Kaam, Stefano Schiavon, Hui Zhang, Edward Arens, Karl Brown, Therese Peffer, Carl Blumstein, et al. 2017. Changing the Rules: Innovative Low-Energy Occupant-Responsive HVAC Controls and Systems. (2017).

[17] Carlo Alberto Boano, Simon Duquennoy, Anna Förster, Omprakash Gnawali, Romain Jacob, Hyung-Sin Kim, Olaf Landsiedel, Ramona Marfievici, Luca Mottola, Gian Pietro Picco, Xavier Vilajosana, Thomas Watteyne, and Marco Zimmerling. 2018. IoTBench: Towards a Benchmark for Low-power Wireless Networking. In *Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench 2018)*.

[18] Michael Buettner, Gary V Yee, Eric Anderson, and Richard Han. 2006. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 307–320.

[19] Philip Buonadonna, Jason Hill, and David Culler. 2001. Active message communication for tiny networked sensors. (2001). https://pdfs.semanticscholar.org/83d1/172d8e6164e40cae65aab28b02a9dac9acef.pdf.

[20] David D Clark. 1985. The Structuring of Systems Using Upcalls. In *ACM SIGOPS Operating Systems Review*, Vol. 19. ACM, 171–180.

[21] Crossbow. [n. d.]. Imote2. ([n. d.]). http://wsn.cse.wustl.edu/images/e/e3/Imote2_Datasheet.pdf.

[22] Ioan Deaconu and Dan Ştefan Tudose. 2017. Sparrow: An Energy Harvesting Wireless Sensor Node. In *Control, Decision and Information Technologies (CoDIT), 2017 4th International Conference on*. IEEE, 0410–0415.

[23] Manjunath Doddavenkatappa, Mun Choon Chan, and Akkihebbal L Ananda. 2011. Indriya: A Low-cost, 3D Wireless Sensor Network Testbed. In *International Conference on Testbeds and Research Infrastructures*. Springer, 302–316.

[24] Adam Dunkels. 2011. The ContikiMac Radio Duty Cycling Protocol. (2011).

[25] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 455–462.

[26] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*. Acm, 29–42.

[27] Simon Duquennoy, Beshr Al Nahas, Olaf Landsiedel, and Thomas Watteyne. 2015. Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH. In *Proceedings of the 13th ACM conference on embedded networked sensor systems*. ACM, 337–350.

[28] Simon Duquennoy, Atis Elsts, Al Nahas, and George Oikonomou. 2017. TSCH and 6TiSCH for Contiki: Challenges, Design and Evaluation. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (IEEE DCOSS)*.

[29] Simon Duquennoy, Olaf Landsiedel, and Thiemo Voigt. 2013. Let the tree bloom: Scalable opportunistic routing with orpl. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2.

[30] Prabal Dutta, Jay Taneja, Jaein Jeong, Xiaofan Jiang, and David Culler. 2008. A Building Block Approach to Sensornet Systems. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, 267–280.

[31] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. 1999. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. ACM, 263–270.

[32] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. 2011. Efficient network flooding and time synchronization with glossy. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE, 73–84.

[33] Chipcon Products from Texas Instruments. 2004. 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. (2004). https://www.ti.com/lit/ds/symlink/cc2420.pdf.

[34] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. 2014. The NesC Language: A Holistic Approach to Networked Embedded Systems. *Acm Sigplan Notices* 49, 4 (2014), 41–51.

[35] Bluetoth Mesh Working Group. 2017. Mesh Profile v1.0. (2017).

[36] Thread Group. [n. d.]. Thread. ([n. d.]). https://threadgroup.org.

[37] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. 2000. System Architecture Directions for Networked Sensors. *ACM SIGOPS operating systems review* 34, 5 (2000), 93–104.

[38] Jason L Hill and David E Culler. 2002. Mica: A wireless platform for deeply embedded networks. *IEEE micro* 22, 6 (2002), 12–24.

[39] Texas Instrument. 2014. SmartRF^{TM} Flash Programmer User Manual. (2014). http://www.ti.com/lit/ug/swru069g/swru069g.pdf.

[40] Texas Instruments. 2002. MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller. (2002). https://www.ti.com/lit/ds/symlink/msp430f1611.pdf.

[41] Texas Instruments. 2015. CC2538 Powerful Wireless Microcontroller System-On-Chip for 2.4-GHz IEEE 802.15.4, 6LoWPAN, and ZigBee Applications. (2015). http://www.ti.com/lit/ds/symlink/cc2538.pdf.

[42] Texas Instruments. 2015. Multi-Standard CC2650 SensorTag Design Guide. (2015). http://datasheet.octopart.com/CC2650STK-Texas-Instruments-datasheet-43919922.pdf.

[43] Texas Instruments. 2016. CC2650 SimpleLink^{TM} Multistandard Wireless MCU. (2016). http://www.ti.com/lit/ds/symlink/cc2650.pdf.

[44] Texas Instruments. 2016. MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers. (2016). http://www.ti.com/lit/ds/symlink/msp430fr5994.pdf.

[45] Davide Italiano and Alexander Motin. 2013. Calloutng: A New Infrastructure for Timer Facilities in the FreeBSD Kernel. (2013).

[46] Jennic. 2008. Sensor Board Reference Manual. (2008). https://www.sparkfun.com/datasheets/Wireless/Zigbee/Jennic-Node-Reference.pdf.

[47] Deokwoo Jung, Zhenjie Zhang, and Marianne Winslett. 2017. Vibration Analysis for IoT Enabled Predictive Maintenance. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE, 1271–1282.

[48] Cheng Kai, Qiu Shuangqing, Wang Xiaolu, and Xu Li. 2016. Warehouse Environment Wireless Monitoring System based on SAMR21 of LwMesh. *Microcontrollers & Embedded Systems* 2 (2016), 007.

[49] Jeonghoon Kang, Jaechul Kim, Du-Hwan Yeo, Jongmin Hyun, Kooklae Jo, Taejoon Choi, Pil-Mhan Jung, Hangyeol Kim, Su Chang Lee, and Sukun Kim. 2012. Modular approach in sensor board design. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*. ACM, 365–366.

[50] Osama U Khan and David D Wentzloff. 2015. 8.1 nJ/b 2.4 GHz Short-range Communication Receiver in 65 nm CMOS. *IEEE Transactions on Circuits and Systems I: Regular Papers* 62, 7 (2015), 1854–1862.

[51] Hyung-Sin Kim, Hongchan Kim, Jeongyeup Paek, and Saewoong Bahk. 2017. Load Balancing Under Heavy Traffic in RPL Routing Protocol for Low Power and Lossy Networks. *IEEE Transactions on Mobile Computing* 16, 4 (2017), 964–979.

[52] Hyung-Sin Kim, Jeonggil Ko, David E Culler, and Jeongyeup Paek. 2017. Challenging the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL): A Survey. *IEEE Communications Surveys & Tutorials* 19, 4 (2017).

[53] Hyung-Sin Kim, Jeongyeup Paek, David E Culler, and Saewoong Bahk. 2017. Do Not Lose Bandwidth: Adaptive Transmission Power and Multihop Topology Control. In *Distributed Computing in Sensor Systems (DCOSS), 2017 13th International Conference on*. IEEE, 99–108.

[54] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. 2007. Health Monitoring of Civil Infrastructures

using Wireless Sensor Networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*. ACM, 254–263.
[55] Kevin Klues, Vlado Handziski, Chenyang Lu, Adam Wolisz, David Culler, David Gay, and Philip Levis. 2007. Integrating concurrency control and energy management in device drivers. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 251–264.
[56] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Razvan Musaloiu-Elefteri, Philip Levis, Andreas Terzis, and Ramesh Govindan. 2009. TOSThreads: Thread-Safe and Non-Invasive Preemption in TinyOS.. In *ACM SenSys*, Vol. 9. 127–140.
[57] JeongGil Ko, Kevin Klues, Christian Richter, Wanja Hofer, Branislav Kusy, Michael Bruenig, Thomas Schmid, Qiang Wang, Prabal Dutta, and Andreas Terzis. 2012. Low Power or High Performance? a Tradeoff Whose Time Has Come (and Nearly Gone). In *European Conference on Wireless Sensor Networks*. Springer, 98–114.
[58] Sam Kumar, Michael P Andersen, Hyung-Sin Kim, and David E. Culler. 2018. Bringing Full-Scale TCP to Low Power Networks. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM.
[59] Branislav Kusy, Christian Richter, Wen Hu, Mikhail Afanasyev, Raja Jurdak, Michael Brünig, David Abbott, Cong Huynh, and Diethelm Ostry. 2011. Radio Diversity for Reliable Communication in WSNs. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE, 270–281.
[60] Taeseop Lee, Jonghun Han, Myung-Sup Lee, Hyung-Sin Kim, and Saewoong Bahk. 2017. CABLE: Connection Interval Adaptation for BLE in Dynamic Wireless Environments. In *Sensing, Communication, and Networking (SECON), 2017 14th Annual IEEE International Conference on*. IEEE, 1–9.
[61] Taeseop Lee, Myung-Sup Lee, Hyung-Sin Kim, and Saewoong Bahk. 2016. A synergistic architecture for RPL over BLE. In *Sensing, Communication, and Networking (SECON), 2016 13th Annual IEEE International Conference on*. IEEE, 1–9.
[62] Philip Levis. 2012. Experiences from a Decade of TinyOS Development.. In *Proc. USENIX OSDI*. 207–220.
[63] Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric A Brewer, and David E Culler. 2004. The Emergence of Networking Abstractions and Techniques in TinyOS.. In *NSDI*, Vol. 4. 1–1.
[64] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An Operating System for Sensor Networks. In *Ambient intelligence*. Springer, 115–148.
[65] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 234–251.
[66] Nicholas D Matsakis and Felix S Klock II. 2014. The Rust Language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.
[67] William P McCartney and Nigamanth Sridhar. 2006. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 167–180.
[68] MEMSIC. [n. d.]. MICAz. ([n. d.]). http://www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf.
[69] David Moss and Philip Levis. 2008. BoX-MACs: Exploiting physical and link layer boundaries in low-power networking. *Computer Systems Laboratory Stanford University* (2008), 116–119.
[70] MoteiV. 2004. Telos Revision B: Humidity, Light, and Temperature sensors with USB. (2004). http://www2.ece.ohio-state.edu/~bibyk/ee582/telosMote.pdf.
[71] Google Nest. [n. d.]. OpenThread. ([n. d.]). https://github.com/openthread/openthread.
[72] Openhardwarefortheinternetofthings. 2017. OpenMote B. (2017). http://www.openmote.com/.
[73] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. 2006. Cross-level Sensor Network Simulation with Cooja. In *Local computer networks, proceedings 2006 31st IEEE conference on*. IEEE, 641–648.
[74] Jaeyeon Park, Woojin Nam, Jaewon Choi, Taeyeong Kim, Dukyong Yoon, Sukhoon Lee, Jeongyeup Paek, and JeongGil Ko. 2017. Glasses for the Third Eye: Improving the Quality of Clinical Data Analysis with Motion Sensor-based Data Filtering. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 8.
[75] Joseph Polastre, Robert Szewczyk, and David Culler. 2005. Telos: Enabling Ultra-Low Power Wireless Research. In *Proceedings of the 4th international symposium on Information processing in sensor networks*. IEEE Press, 48.
[76] Bluetooth SIG Proprietary. 2016. Bluetooth Specification Version 5.0. (2016).
[77] John Regehr, Alastair Reid, and Kirk Webb. 2005. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems (TECS)* 4, 4 (2005), 751–778.
[78] Christopher M Sadler and Margaret Martonosi. 2006. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 265–278.
[79] Thomas Schmid, Roy Shea, Mani B Srivastava, and Prabal Dutta. 2010. Disentangling Wireless Sensing from Mesh Networking. In *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors*. ACM, 3.
[80] Markus Schuß, Carlo Alberto Boano, and Kay Römer. 2018. Moving Beyond Competitions: Extending D-Cube to Seamlessly Benchmark Low-Power Wireless Systems. In *Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench 2018)*.
[81] Nordic Semiconductor. [n. d.]. nRF52840 Product Specification v1.0. ([n. d.]). http://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.0.pdf.
[82] Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. 2017. Secure Sharing of Partially Homomorphic Encrypted IoT Data. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 29.
[83] Wentao Shang, Alex Afanasyev, and Lixia Zhang. 2016. The Design and Implementation of the NDN Protocol Stack for RIOT-OS. In *Globecom Workshops (GC Wkshps), 2016 IEEE*. IEEE, 1–6.
[84] Suresh Siddha, Venkatesh Pallipadi, and AVD Ven. 2007. Getting Maximum Mileage out of Tickless. In *Proceedings of the Linux Symposium*, Vol. 2. Citeseer, 201–207.
[85] Michael Spörk, Carlo Alberto Boano, Marco Zimmerling, and Kay Römer. 2017. BLEach: Exploiting the Full Potential of IPv6 over BLE in Constrained Embedded IoT Devices. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 2.
[86] Felix Sutton, Marco Zimmerling, Reto Da Forno, Roman Lim, Tonio Gsell, Georgia Giannopoulou, Federico Ferrari, Jan Beutel, and Lothar Thiele. 2015. Bolt: A stateful processor interconnect. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. ACM, 267–280.
[87] Federico Terraneo, Alberto Leva, and William Fornaciari. 2016. A High-Performance, Energy-Efficient Node for a Wide Range of WSN Applications.. In *EWSN*. 241–242.
[88] Saurabh Verma and Ashima S Dabare. 2007. Understanding Clock Domain Crossing Issues. *EE Times* (2007).
[89] Xavier Vilajosana, Pere Tuset, Thomas Watteyne, and Kris Pister. 2015. OpenMote: Open-Source Prototyping Platform for the Industrial IoT. In *International Conference on Ad Hoc Networks*. Springer, 211–222.
[90] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. 2003. Why Events Are a Bad Idea (for High-Concurrency Servers).. In *HotOS*. 19–24.
[91] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer SJ Pister. 2001. Smart Dust: Communicating with a Cubic-millimeter Computer. *Computer* 34, 1 (2001), 44–51.
[92] Brad Wheeler, Filip Maksimovic, Nima Baniasadi, Sahar Mesri, Osama Khan, David Burnett, Ali Niknejad, and Kris Pister. 2017. Crystal-free Narrow-band Radios for Low-cost IoT. In *Radio Frequency Integrated Circuits Symposium (RFIC), 2017 IEEE*. IEEE, 228–231.
[93] Heiko Will, Kaspar Schleiser, and Jochen Schiller. 2009. A Real-time Kernel for Wireless Sensor Networks Employed in Rescue Scenarios. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*. IEEE, 834–841.
[94] Zolertia. 2010. Z1 Datasheet 1.1. (2010). http://zolertia.sourceforge.net/wiki/images/e/e8/Z1_RevC_Datasheet.pdf.