Proceedings of the ASME 2015 International Mechanical Engineering Congress & Exposition
IMECE2015
November 13-19, 2015, Houston, Texas

IMECE2015-50173

ENABLING RIGHT-PROVISIONED MICROPROCESSOR ARCHITECTURES FOR THE INTERNET OF THINGS

Tosiron Adegbija
Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ, USA
tosiron@email.arizona.edu

Anita Rogacs
Hewlett-Packard (HP) Laboratories
Palo Alto, CA, USA
rogacs@hp.com

Chandrakant Patel
Hewlett-Packard (HP) Laboratories
Palo Alto, CA, USA
chandrakant.patel@hp.com

Ann Gordon-Ross[1]
Department of Electrical and Computer Engineering
University of Florida, Gainesville, FL, USA
ann@ece.ufl.edu

ABSTRACT

The Internet of Things (IoT) consists of embedded low-power devices that collect and transmit data to centralized head nodes that process and analyze the data, and drive actions. The proliferation of these connected low-power devices will result in a data explosion that will significantly increase data transmission costs with respect to energy consumed and latency. *Edge computing* performs computations at the edge nodes prior to data transmission to interpret and/or utilize the data, thus reducing transmission costs. In this work, we seek to understand the interactions between IoT applications' execution characteristics (e.g., compute/memory intensity, cache miss rates, etc.) and the edge nodes' microarchitectural characteristics (e.g., clock frequency, memory capacity, etc.) for efficient and effective edge computing. Thus, we present a broad and tractable IoT application classification methodology and using this classification, we analyze the microarchitectural characteristics of a wide range of state-of-the-art embedded system microprocessors and evaluate the microprocessors' applicability to IoT computation using various evaluation metrics. We also investigate and quantify the impact of leakage power reduction on the overall energy consumption across different architectures. Our work provides insights into the microarchitectural characteristics' impact on system performance and efficiency for various IoT application requirements. Our work also provides a foundation for the analysis and design of a diverse set of microprocessor architectures for IoT edge computing.

INTRODUCTION AND MOTIVATION

The Internet of Things (IoT) refers to a pervasive presence of a variety of devices that offer connectivity, systems, and services that spans a variety of protocols, domains, and applications. The goal of the IoT is to reduce reliance on human intervention for data acquisition, interpretation, and use. The IoT has been described as one of the disruptive technologies that will transform life, business, and the global economy [23]. Based on analysis of key potential IoT use-cases (e.g., healthcare, smart

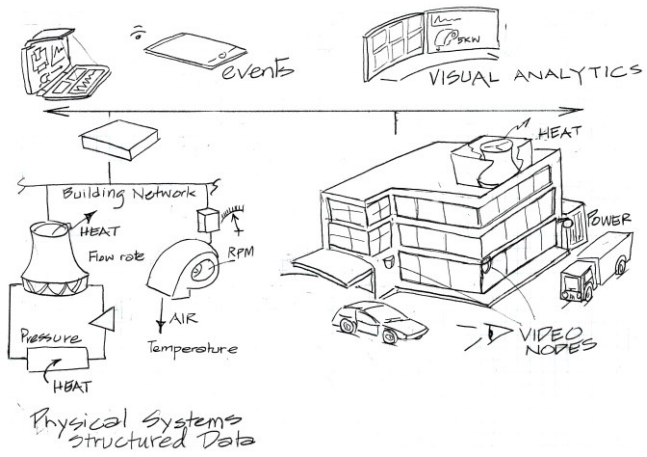


Figure 1. Components of the Internet of Things (IoT)

[1]*Also affiliated with the NSF Center for High-Performance Reconfigurable Computing (CHREC) at UF*

cities, smart home, transportation, manufacturing, etc.), Gartner Technology Research [10] has estimated that by 2020, the IoT will constitute a trillion dollar economic impact and include 26 billion low-power devices that will generate massive amounts of data.

Figure 1 depicts the high-level components in a traditional IoT model. The traditional IoT comprises of several low-power/low-performance *edge nodes,* such as sensor nodes, that gather data and transmit the data to high-performance head nodes, such as servers, that perform computations for visualization and analytics. In Figure 1, data aggregation from edge nodes in a building facilitates power and cooling management. Data centers are a prime example of this IoT application domain [5][25].

The growth of the IoT and the resulting exponential increase in acquired/transmitted data poses significant bandwidth and latency challenges. These challenges are exacerbated by the intrinsic resource constraints of most embedded edge nodes, coupled with increasing consumer demand for high-performance applications, resulting in more complex data. For example, a closed-circuit television (CCTV) camera surveillance system can be used to acquire (sense) high-resolution images and video streams for security analysis in high-population areas (e.g., a sports arenas could be the building in Figure 1) to detect the presence of persons-of-interest (POIs). For a fine-grained, more complete coverage, the surveillance system should be scalable to a network of many cameras, however this scalability poses challenges for the traditional IoT since large amounts of data must be transmitted over bandwidth-limited networks to process the acquired data. Furthermore, transmitting acquired data to a head node for processing poses additional challenges for real-time systems where the latency must adhere to deadline constraints.

Concomitant to these bandwidth and latency challenges, the traditional IoT model can also result in significant energy overhead. Previous work [17] established that energy consumed while transmitting data is significantly more than the energy consumed while performing computations on the data. For example, the energy required by Rockwell Automation's sensor nodes to transmit one bit of data requires 1500-2000X more energy than executing a single instruction (depending on the transmission range and specific computations) [26].

To address these challenges, *edge computing* performs computations that process, interpret, and use data at the edge node, in order to minimize the transmitted data. Using our CCTV example, rather than acquiring images and video streams at the cameras and transmitting this data to a head node for computation, edge computing could include face recognition at the level of the CCTV cameras, such that only information about the presence of POIs is transmitted to the head node, thus significantly reducing data transmission cost.

Since transmission energy dominates computation energy in traditional edge nodes, edge computing has the potential to significantly reduce overall energy consumption in edge nodes. Gaura et al. [11] showed that edge computing, otherwise referred to as *edge mining*, can quantifiably reduce the amount of transmitted data, thereby reducing transmission energy and remote storage requirements. However, to support edge computing, especially in data-rich use-cases, the edge nodes' computing capabilities must be sufficient to perform and sustain the required computations.

To ensure that architectures designed for the IoT have sufficient computing capabilities, tradeoffs in architectural characteristics should be determined and considered when designing an IoT device or when selecting the IoT devices' system configurations (e.g., cache size, clock frequency, etc.). However, due to the wide variety of IoT applications and the diverse set of available architectures, determining the appropriate architectures is challenging. This paper seeks to address these challenges and motivate future research in this direction.

Most current IoT edge nodes consist of communication systems (e.g., Bluetooth [14], Wi-Fi [19], Zigbee [21], etc.), sensors (e.g., temperature, pressure, etc.), actuators, resource-constrained energy/power sources (e.g., batteries, solar panels, etc), low-power/low-performance processor (e.g., microcontroller units, etc.), and memory. In this work, we focus on the edge node's processing component and seek to understand the microarchitectural characteristics that are required to support IoT edge computing. In order to understand the architectures that will support IoT edge computing, we must first understand the applications that will execute on those architectures and the applications' characteristics. Since most state-of-the art IoT devices consist of microcontroller units (MCU) with minimal computational capabilities, our goal is to determine if the current MCUs are sufficient for the computing capabilities required by emerging IoT applications and to propose solutions to satisfy the IoT applications' requirements.

In this paper, we perform an expansive study and characterization of the emerging IoT application space and propose an application classification methodology to broadly represent IoT applications. Based on this classification, we propose a benchmark suite that provides a tractable way to represent key computations that occur on the IoT application space. This methodology is based on *computational dwarfs*, as proposed by Asanovic et al. [2], which allows representation of computational patterns at a high level of abstraction, and has been extended by other previous work to different computing domains (e.g., [22]). Furthermore, we propose a high-level methodology for identifying right-provisioned architectures for edge computing use-cases, based on the executing applications and the applications' execution characteristics (e.g., compute intensity, memory intensity, etc.). Using this methodology, we study several state-of-the-art low power processors' microarchitectural characteristics and evaluate these processors' applicability to perform IoT edge computing using various performance metrics. Since leakage power contributes significantly to the overall energy consumption in low-power devices, we investigate and quantify the impact of power optimization on overall energy consumption. Using power gating [15] as a power optimization example, we show that the choice of processors equipped with power optimization mechanisms to reduce leakage power is dependent on the proportion of time during which an application is executing—the application's *duty cycle*. Thus, to maximize the power optimization potential, the processors must be carefully selected with respect to the executing applications.

RELATED WORK

Due to the expected growth of the IoT, an increasing amount of research [4][12][13] focuses on understanding and discovering insights into various aspects of the IoT. Much emphasis has been placed on the software layer of the IoT, however, the edge nodes' hardware components and processing capabilities must also be considered [27], especially in the context of edge computing.

Bonomi et al. [7] proposed *fog computing* as a virtualized platform that provides compute, storage, and networking services between edge nodes and cloud computing data centers. Fog computing reduces the bandwidth bottleneck and latency by moving computation closer to the edge nodes. Our work explores further reduction in bandwidth, latency, and energy consumption by equipping the edge nodes with sufficient computation capacity in order to minimize data transmission. Gaura et al. [11] examined the benefits of edge mining, in which data mining takes place on the edge devices. The authors showed that edge mining has the potential to reduce the amount of transmitted data, thus reducing energy consumption and storage requirements.

Previous works have proposed classifications for various IoT components. Gubbi et al. [13] presented a taxonomy for a high level definition of IoT components with respect to hardware, middleware, and presentation/data visualization. Tilak et al. [30] presented a taxonomy to classify wireless sensor networks according to different communication functions, data delivery models, and network dynamics. Tory et al. [31] presented a high level visualization taxonomy that classified algorithms based on the characteristics of the data models. However, to the best of our knowledge, our work presents the first classification of IoT applications based on the applications' functions. Since the applications' functions determine the execution characteristics, the functions have a more direct impact on the microprocessor requirements.

IOT APPLICATION CLASSIFICATION

The IoT offers computing potential for many application domains, including transportation and logistics, healthcare, smart environment, personal and social domains [4], etc. Since it is impractical to consider every IoT application within these domains, we perform an expansive study of IoT use-cases and the application functions performed by these use-cases. We propose an application classification methodology that provides a high level, broad, and tractable description of a variety IoT applications. Our IoT application classification consists of five application functions: *sensing, communications, image processing, compression (lossy/lossless), security,* and *fault tolerance*. In this section, we describe the application functions and motivate these functions using specific examples of current and/or emerging IoT applications.

Sensing

Sensing involves data acquisition (e.g., temperature, pressure, motion, etc.) about objects or phenomena, and is increasingly common in several application domains. In these applications, activities/information/data of interest are gathered for further processing and decision making. We use sensing in our IoT application classification to represent applications where data acquired using sensors must be converted to a more useable form. Our motivating example for sensing applications is *sensor fusion* [24], where sensed data from multiple sensors are fused to create data that is considered qualitatively or quantitatively more accurate and robust than the original data.

Sensor fusion algorithms can involve various levels of complexity and compute/memory intensity. For example, sensor fusion could involve aggregating data from various sources using simple mathematical computations, such as addition, minimum, maximum, mean, etc. Alternatively, sensor fusion could involve more computationally complex/expensive applications, such as fusing vector data (e.g., video streams from multiple sources), which requires a substantial increase in intermediate processing.

Communications

Communications is one of the most common IoT application functions due to the IoT's intrinsic connected structure, where data transfers traverse several connected nodes. There are many communication technologies (e.g., Bluetooth, Wi-Fi, etc.), and communication protocols (e.g., transfer control protocol (TCP), the emerging *6lowpan* (IPv6 over low power wireless personal area network), etc.). However, in this work, we highlight *software defined radio (SDR)* [18], which is a communication system in which physical layer functions (e.g., filters, modems, etc.) that are typically implemented in hardware are implemented in software.

SDR is an emerging communication system because of SDR's inherent flexibility, which allows for flexible incorporation and enhancements of multiple radio functions, bands, and modes, without requiring hardware updates. SDR typically involves an antenna, an analog-to-digital converter (ADC) connected to an antenna (for receiving) and a digital to analog converter (DAC) connected to the antenna (for transmitting). Digital signal processing (DSP) operations (e.g., Fast Fourier Transform (FFT)) are then used to convert the input signals to any form required by the application. SDR applications are typically compute intensive, with small data and instruction memory footprints. Other examples of communication applications include packet switching and TCP/IP.

Image processing

We use image processing to represent applications that involve any form of signal processing where the input is an image or video stream from which characteristics/parameters must be extracted/identified, or the input must be converted to a more usable form. Several IoT applications, such as automatic number license plate recognition, traffic sign recognition, face recognition, etc., involve various forms of image processing. For example, face recognition involves operations, such as face detection, landmark recognition, feature extraction, and feature classification, all of which involve image processing.

Several image processing use-cases and applications are still nascent, and are expected to grow significantly in the coming years [1]. These applications typically require significant computation capabilities, since image processing involves compute-intensive operations, such as matrix multiplications. Furthermore, some image processing applications require large input, intermediate, or output data to be stored (e.g., medical imaging), thus requiring a large amount of memory storage.

TABLE 1. MICROARCHITECTURE CONFIGURATIONS (I=INSTRUCTIONS, D=DATA, L1 = LEVEL ONE, L2=LEVEL TWO)

	Conf1	*Conf2*	*Conf3*	*Conf4*
Sample CPU	ARM Cortex M4	Intel Quark	ARM Cortex A7	ARM Cortex A15
Frequency	48 MHz	400 MHz	1 GHz	1.9 GHz
Number of cores	1	1	4	4
Pipeline stages	3	5	8	15
Cache	None	None	32 KB I/D L1, 1MB L2	32 KB I/D L1, 2MB L2
Memory	512 KB flash	2 GB RAM	2 GB support	1 TB RAM support
Execution	In-order	In-order	In-order	Out-of-order

Compression

With the increase in data and bandwidth-limited systems, compression can reduce communication requirements to ensure that data is quickly retrieved, transmitted, and/or analyzed. Additionally, since most IoT devices are resource-constrained, compression also reduces storage requirements when storage on the edge node is required.

Compression techniques can be broadly classified as *lossy* or *lossless* compression. Lossy compression (e.g., JPEG) typically exploits the perceptibility of the data in question, and removes unnecessary data, such that the lost data is imperceptible to the user. Alternatively, lossless compression removes statistically redundant data in order to concisely represent data. Lossless compression typically achieves a lower compression ratio and is usually more compute and memory intensive than lossy compression. However, lossy compression may be unsuitable in some scenarios where high data fidelity is required to maintain the quality of service (QoS) (e.g., in medical imaging).

Security

Since IoT devices are often deployed in open environments, where the devices are susceptible to malicious attacks, security applications are necessary to maintain the integrity of both the devices and the data. Furthermore, sensitive scenarios (e.g., medical diagnostics) may require security applications to prevent unauthorized access to sensitive data.

We highlight data encryption [28], which is a common technique for ensuring data confidentiality, wherein an encryption algorithm is used to generate encrypted data that can only be read/used if decrypted. Data encryption applications (e.g., secure hash algorithm) are typically compute intensive and memory intensive, since encryption speed is also dependent on the memory access latency for data retrieval and storage.

Fault tolerance

Fault tolerance refers to a system's ability to operate properly in the event of a failure of some of the system's components. Fault tolerant applications are especially vital since IoT devices may be deployed in harsh and unattended environments, where QoS must be maintained in potentially adverse conditions, such as cryogenic to extremely high temperatures, shock, vibration, etc.

Fault tolerance can be hardware-based, such as hardware-based RAID (redundant array of independent disks), which are storage devices that use redundancy to provide fault tolerance (we note that software-based methods do exist but typically suffer from reduced reliability). Alternatively, software-based fault tolerance involves applications and algorithms that perform operations, such as memory scrubbing, cyclic-redundancy checks, error detection and correction, etc.

IOT MICROARCHITECTURE CONFIGURATIONS

We performed an extensive study of the state-of-the-art in commercial-off-the-shelf (COTS) embedded systems microprocessor architectures from several designers and manufacturers ranging from low-end microcontrollers to high-end/high-performance low-power embedded systems microprocessors. Based on publicly available information on these processors' configurations and interactions with engineers directly involved with processor design with different manufacturers, we categorized the microprocessors in terms of several device characteristics, including number of cores, on-chip memory (e.g., cache), off-chip memory support, power consumption, number of pipeline stages, etc. Using this information, we developed a set of potential microarchitecture configurations for IoT edge computing support. These configurations represent the range of available state-of-the-art COTS microprocessors. Note that microprocessors could include central processing units (CPUs), graphics processing units (GPUs), DSPs, etc., however, in our work, we focus on CPUs and intend to evaluate other kinds of microprocessors for future work.

Table 1 details the microarchitecture configurations, comprising of four configurations: *conf1, conf2, conf3,* and *conf4,* representing different kinds of systems. We highlight specific state-of-the-art microcontroller/microprocessor examples to motivate the configurations, however, we note that these configurations are only representative and not necessarily descriptive.

Conf1 represents low-power and low-performance microcontroller units, such as the ARM Cortex-M4 [8] found in several IoT-targeted MCUs from several developers, including Freescale Semiconductors [9], Atmel [3], and STMicroelectronics [29]. *Conf1* contains a single core with 48 MHz clock frequency, three pipeline stages, in-order execution, and support for 1 MB of flash memory.

Conf2 represents recently-developed IoT-targeted CPUs, such as the Intel Quark Technology [16], and contains a single core with 400 MHz clock frequency, five pipeline stages, in-order execution, 16 KB level one (L1) instruction and data caches, and support for 2 GB RAM.

Conf3 represents mid-range CPUs, such as the ARM Cortex-A7 found in several general purpose embedded systems, and contains four cores with 1 GHz clock frequency, 8 pipeline

TABLE 2. APPLICATION FUNCTIONS, REPRESENTATIVE BENCHMARKS, AND BENCHMARK DESCRIPTIONS

Application function	Benchmarks	Benchmark description
Sensing	*matrixTrans* (*_128, _256, _512, _1024*)	Dense matrix transpose of $n \times n$ matrix
Communications	*fft* (*_small* and *_large*)	Fast Fourier Transform (FFT)
Image processing	*matrixMult* (*_128, _256, _512*)	Dense matrix multiplication of $n \times n$ matrix
Lossy compression	*jpeg* (*_small* and *_large*)	Joint Photographic Experts Group (JPEG) compression
Lossless compression	*lz4* (*_mr* and *_xray*)	Lossless data compression
Security	*sha* (*_small* and *_large*)	Secure hash algorithm
Fault tolerance	*crc* (*_small* and *_large*)	Cyclic redundancy check

stages, in-order execution, 32 KB L1 instruction and data caches, 1 MB level two (L2) cache, and support for 2 GB RAM.

Finally, *conf4* represents high-end/high-performance embedded systems CPUs, such as the ARM Cortex-A15, and contains four cores with 1.9 GHz clock frequency, 8 pipeline stages, 32 KB L1 instruction and data caches, 2 MB L2 cache, support for 4 GB RAM, and out-of-order execution. Out-of-order execution allows instructions to execute as soon as the instruction becomes available, unlike in-order execution where instructions must execute in program order.

EXPERIMENTAL METHODOLOGY

This section describes the simulators, our IoT benchmark suite, and the performance metrics considered in this study.

Simulators

To evaluate the applicability of our microarchitecture configurations to the IoT, we used the GEM5 simulator [6] to generate execution statistics while running several benchmarks on the configurations as shown in Table 1. We used the McPAT simulator [20] to generate leakage, dynamic power, and area values for the different configurations, and used Perl scripts to drive our simulations.

Benchmarks and performance metrics

To facilitate our study, we created a benchmark suite based on our application classification methodology with seven kernels to represent emerging IoT edge computing applications. We use the kernels as computational basic blocks to represent the applications' functions, which disconnects the executions from specific implementations, programming languages, and algorithms. This methodology is supported by the concept of *computational dwarfs*, which was introduced by Asanovic et al. [2]. Computational dwarfs represent patterns of computation at high levels of abstractions to encompass several computational methods in modern computing. Within these dwarfs, kernels are used to expose computational nuances that reveal characteristics that may not be visible at the level of the dwarfs.

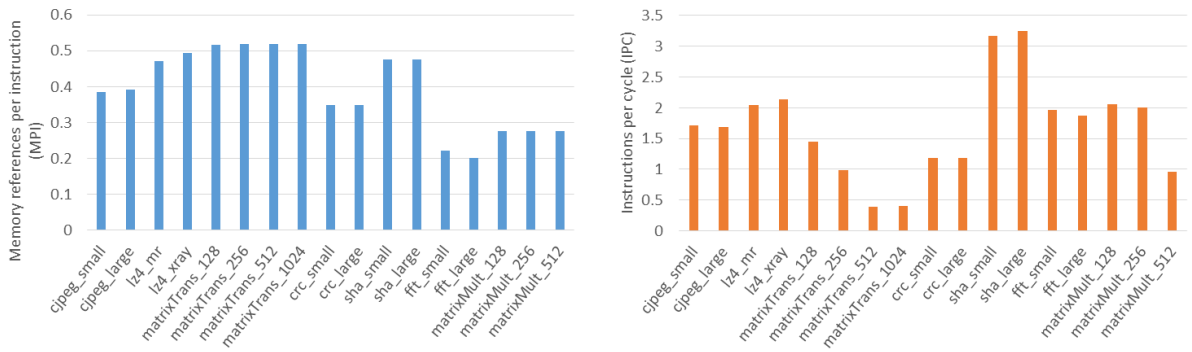
Table 2 depicts our application functions, each application function's representative benchmarks, and the benchmarks' descriptions. For each benchmark, we used different input data sizes to model different real-world usage scenarios, and cross-compiled all benchmarks for the ARM instruction set architecture (ISA). We omit the detailed descriptions of the benchmarks for brevity.

To quantitatively compare the microarchitecture configurations, we use *execution time*, *energy*, *performance* measured in giga operations per second (GOPS), and *efficiency* measured in performance per watt (GOPS/W).

RESULTS

In this section, we present simulation results for execution time, energy, performance, and performance per watt on the microarchitecture configurations listed in Table 1. We also perform sensitivity analysis with respect to varying application data sizes, various microarchitectural characteristics, and evaluate the impacts of idle energy and leakage power reduction.

We note that in this work we did not explore the impact of multiple cores on parallelizable applications. All simulations were performed using the single core versions of the microarchitecture configurations shown in Table 1, and we leave multi-core exploration for future work.



(a) *(b)*

Figure 2. (a) Memory references per instruction (MPI) and (b) instructions per cycle (IPC) of the benchmarks with different data sizes

Execution characteristics and sensitivity to data sizes

To evaluate the execution characteristics of the different benchmarks, we used the percentage of memory references per instruction (MPI) and the instructions per cycle (IPC) to provide insight into the benchmarks' memory and compute intensities, respectively. Note that IPC can also provide an indication of a benchmark's memory intensity (e.g., a low IPC could indicate long memory access times due to accesses to lower level memory, and hence, a memory intensive benchmark).

First, we evaluated the MPI and IPC on *conf4*. While the MPI is microarchitecture-independent (i.e., the MPI remains the same across different microarchitectures), the IPC is microarchitecture-dependent. However, we observed that the IPC also remained relatively stable for different data sizes across all of the configurations. The execution characteristics (MPI and IPC) and sensitivity to different data sizes provides insights into the right provisioning of memory (cache) sizes and/or clock frequencies in order to satisfy the execution requirements.

Figure 2 (a) and (b) depict the MPI and IPC for all of the benchmarks for different input data sizes. Figure 2 (a) shows that the memory intensity for the different benchmarks remained stable regardless of the data size. *matrixTrans* was the most memory intensive benchmark with an MPI of 52%, since most of the computations were performed in memory. Similarly, *sha, cjpeg* and *lz4* were also memory intensive benchmarks with MPIs of up to 49%, while *fft* was the least memory intensive with an MPI of 21%.

Figure 2 (b) shows that the IPCs for different benchmarks were also relatively stable for different data sizes, since the working set sizes for these benchmarks remained stable, except for *matrixTrans* and *matrixMult,* which had variable working set sizes with different data sizes. For example, *matrixTrans'* IPC reduced by 32% and 60% when the data size increased from *matrixTrans_128* to *matrixTrans_256* and from *matrixTrans_256* to *matrixTrans_512*, respectively. The IPC increased because the working set size increased as the data size increased. Thus, there were more processor stalls due to the increased memory activity. However, the working set size remained stable from *matrixTrans_512* to *matrixTrans_1024*, thus, the IPC also remained stable.

Execution time, energy, performance, and efficiency

Figure 3 (a) and (b) depict the execution time and energy of *conf1, conf2,* and *conf3* normalized to *conf4* for all of the benchmarks. We used *conf4* as the base configuration for comparison since this configuration was the biggest of our microarchitecture configurations. Figure 3 (a) shows that *conf1, conf2,* and *conf3* increased the average execution time by 202x, 23x, and 9x, respectively, for all of the benchmarks. These results show that *conf4* outperforms the other configurations when considering latency. Similarly, Figure 3 (b) shows that *conf1, conf2,* and *conf3* increased the energy consumption by 35x, 4.6x, and 4.7x. *Conf4*'s low energy consumption compared to the other configurations was due to the significant reduction in execution time, while the smaller configurations resulted in considerably longer execution times than *conf4*. The graphs do not show *conf1* results for some benchmarks because *conf1*'s memory was too small for the working set size of those benchmarks, and thus the those benchmarks could not be executed on *conf1*. Since *conf1* represents current MCUs that are used on the IoT, our results indicate that these current MCUs are not sufficiently equipped for all edge computing requirements.

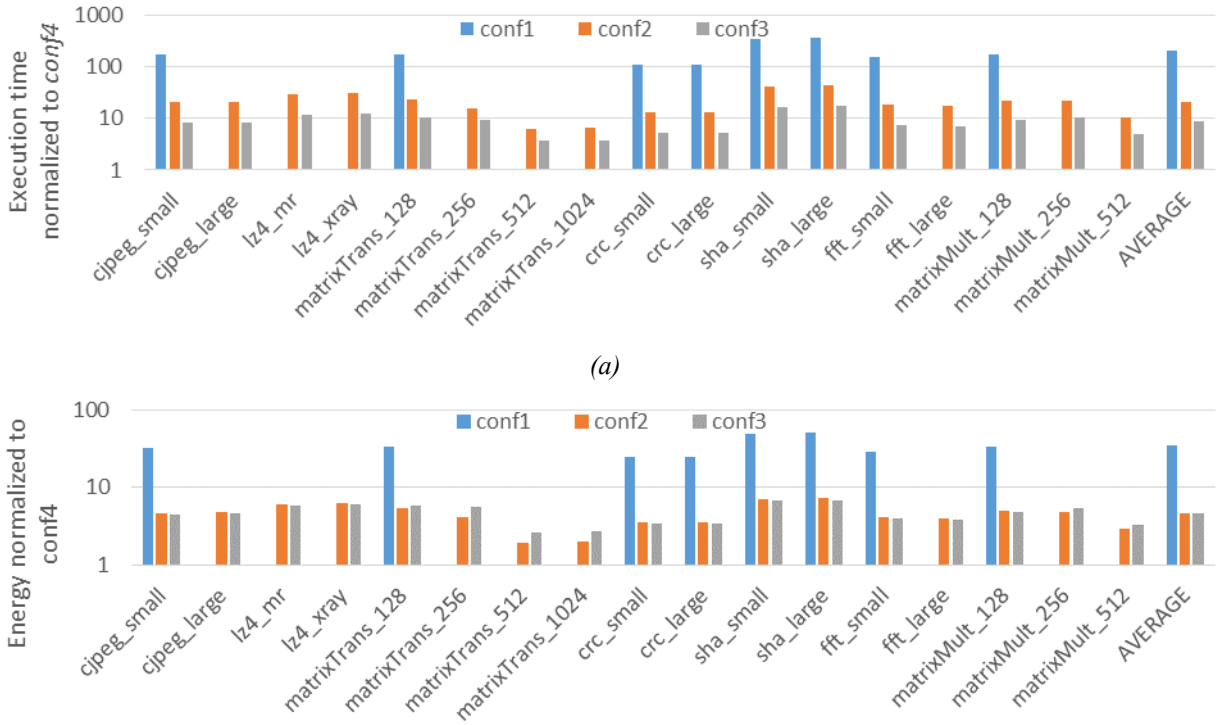


Figure 3. (a) Execution time and (b) energy normalized to *conf4*

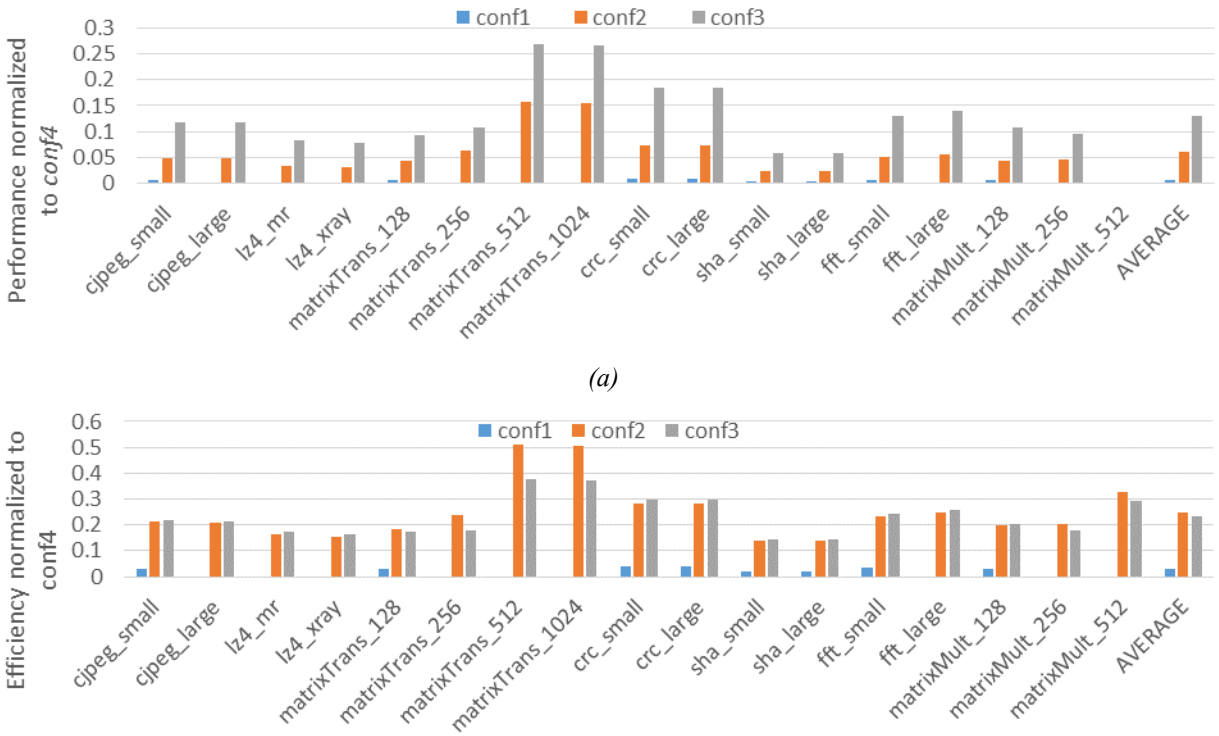


Figure 4. (a) Performance and (b) efficiency normalized to *conf4*

Figure 4 (a) and (b) depict the performance and efficiency of *conf1, conf2,* and *conf3* normalized to *conf4* for all of the benchmarks. Results reveal that c*onf1, conf2,* and *conf3* degraded the performance by 171x, 17x, and 8x, respectively. Compared to *conf4*, *conf1* degraded the efficiency by 33x, while *conf2* and *conf3* degraded the efficiency by 4x. These results reveal the significant improvements achieved by using the larger configurations. In a system that is not energy constrained (e.g., an IoT device that is consistently connected to a power source) or in real-time systems, where minimizing latency is the goal, *conf4* provides the best performance for the system.

Sensitivity to various microarchitectural characteristics

To identify the most impactful microarchitectural characteristics on system execution time, energy, performance, and efficiency, we evaluated *conf4* with a 1 GHz clock frequency, in-order execution, and a 16 KB cache size. For each of these evaluated configurations, all of the other configurations were held constant to isolate the impact of the evaluated configurations. For brevity, we only show results for a subset of the benchmarks' input data sizes, however, all of the benchmarks are included in the averages.

Figure 5 (a) and (b) depict the execution time, energy, performance, and efficiency of *conf4* with a 1 GHz clock frequency normalized to *conf4* with a 1.9 GHz clock frequency. Figure 5 (a) illustrates the significant impact of the clock frequency on execution time and energy consumption. On average over all of the benchmarks, reducing the clock frequency to 1 GHz increased the execution time and energy by 75% and 41%, respectively. However, for *matrixTrans,* reducing the clock frequency to 1 GHz did not change the execution time and reduced the energy by 4%. Since *matrixTrans* was the most memory intensive benchmark and spent more execution time in

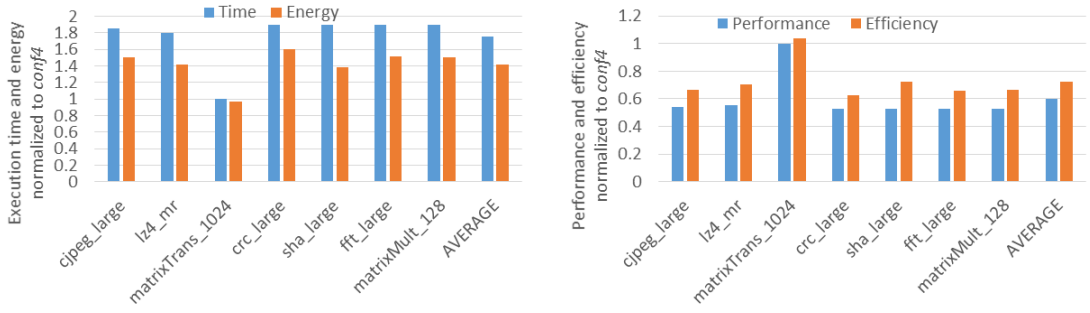
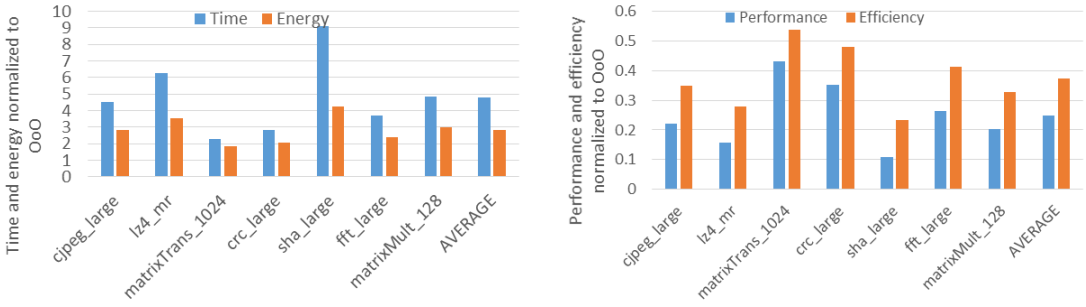


Figure 5. (a) Execution time and energy (b) Performance and efficiency of 1 GHz clock frequency normalized *conf4* (1.9 GHz)



(a) *(b)*

Figure 6. (a) Execution time and energy (b) performance and efficiency of in-order execution normalized to *conf4* (out-of-order)

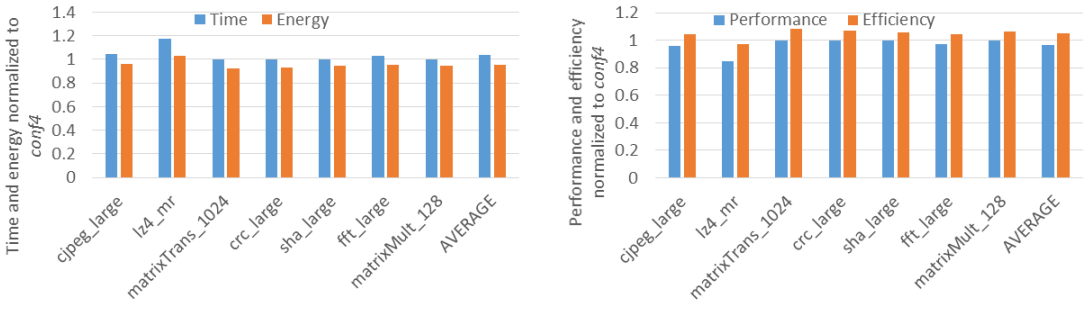
memory activities, reducing the clock frequency had little impact on *matrixTrans* than on the other benchmarks. Figure 5 (b) reveals that reducing the clock frequency reduced the average performance and efficiency by 40% and 27%, respectively. Similarly to the execution time and energy results, for *matrixTrans*, the performance did not change and the efficiency increased by 4% since *matrixTrans's* performance was more dependent on the memory than on the clock frequency. These results show the significant impact that the frequency has on edge computing for IoT applications.

Figure 6 (a) and (b) depict the execution time, energy, performance, and efficiency of *conf4* with in-order execution normalized to *conf4* with out-of-order execution. Figure 6 (a) shows that in-order execution increased the average execution time and energy by 4.8x and 2.9x, respectively, and by as high as 9x for *sha_large*, which is a highly compute intensive and memory intensive benchmark. The results reveal that out-of-order execution provides greater advantages over in-order execution for applications that are compute intensive, however, the impact is reduced for memory intensive applications. Figure 6 (b) shows that in-order execution reduced the average performance and efficiency by 75% and 63%, respectively. Similarly to the execution time and energy, the performance degradation was more significant for the more compute intensive benchmarks, such as *sha_large*.

Figure 7 (a) and (b) depict the execution time, energy, performance, and efficiency of *conf4* when the cache size was reduced to 16 KB normalized to *conf4* with the 32 KB cache. Unlike with the clock frequency and execution order, reducing the cache size did not significantly impact the overall results. Figure 7 (a) shows that the 16 KB cache only increased the average execution time by 4%, with increases as high as 18% for *lz4_mr*. The execution time increased for *lz4_mr* because the 16 KB cache was not large enough to hold *lz4_mr's* working set size, thus incurring cache misses and requiring the data to be fetched from main memory. However, reducing the cache size did not negatively impact the execution time for most of the applications. The 16 KB cache reduced the average energy consumption by 4%, but increased *lz4_mr's* energy consumption by 3% due to the additional cache misses incurred by the 16 KB cache. Similarly, Figure 7 (b) shows that the 16 KB cache degraded the average performance by 3% and improved the average efficiency by 5%. For applications with large working set sizes, such as lossless (*lz4_mr*) and lossy compression (*cjpeg_large*), the 32 KB cache was more appropriate. For all other applications, the 16 KB cache size was sufficient.

Impact of idle energy and power optimization

To illustrate the impact of the idle energy on overall energy consumption, we simulated various application execution scenarios for the shortest and longest running benchmarks (*matrixTrans_128* and *crc_large*). We calculated the total energy consumed as the sum of the energy consumed during application execution and the idle energy, where the idle energy is the product of the leakage power and the idle time. We assumed power gating [15] for power optimization in our evaluations, where the leakage power is reduced by 95%. Power gating is a technique used to reduce a circuit's leakage power consumption by shutting off blocks of the circuit that are not being used. To represent a real-world scenario, we experimented with periodic times and random application execution time intervals, and observed that the results were independent of the periodicity or randomness of the application executions. Thus, we present the



(a) *(b)*

Figure 7. (a) Execution time and energy (b) performance and efficiency of 16 KB cache normalized to *conf4* (32 KB cache)

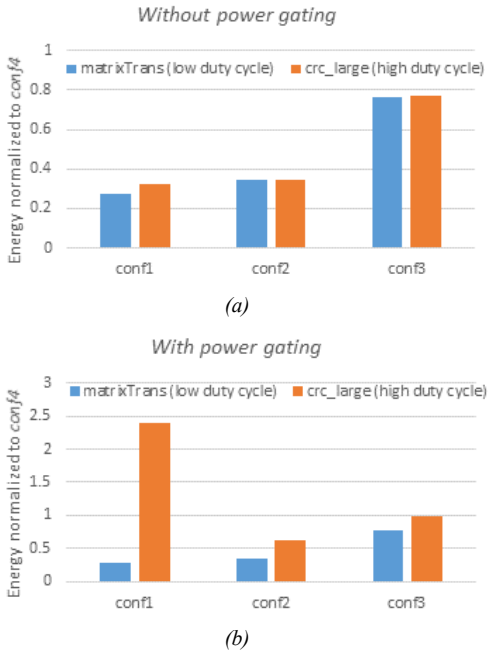


Figure 8. Total energy consumption (including idle energy) of *conf1, conf2,* and *conf3* normalized to *conf4* (a) without power gating, and (b) with power gating

results for both periodic and random execution time intervals together in this subsection.

Figure 8 (a) and (b) depict the total energy consumed by *conf1, conf2,* and *conf3* normalized to *conf4* for *matrixTrans* and *crc_large,* representing low and high duty cycle applications, respectively, without power gating and with power gating. Figure 8 (a) shows that without power gating, *conf1* consumed the lowest amount of energy for both the low and high duty cycle benchmarks. Even though *conf3* and *conf4* executed the applications fastest and accounted for the least dynamic energy consumption, both configurations had high leakage power, thus negating the energy savings from the short execution times. Figure 8 (b) shows similar results for the *matrixTrans* with power gating. Since *matrixTrans* is a short benchmark that executed relatively fast on all of the configurations, there was not enough difference in the configurations' idle times for power gating to provide any significant benefit. However, for *crc_large, conf1* consumed the most overall energy with power gating, while *conf2* consumed the least energy. Due to the length of the application, *conf1* spent most of the time executing the application, while *conf2, conf3,* and *conf4* were able to go into the idle mode much faster due to faster execution times. *Conf2* provided the optimal balance, across all of the configurations, between the time spent executing the application and the time spent idling. Thus, these results reveal that the benefits of power gating are dependent on the application's duty cycle. Applications executing on configurations that have low duty cycles have a higher potential of benefiting from power gating.

To further evaluate the impact of idle energy, we considered a scenario in which multiple applications were randomly executed periodically or at random time intervals (we have omitted the figures for brevity). Without power gating, *conf2* consumed the lowest energy on average, showing that this configuration provided a good balance between idle time and leakage power. However, with power gating, *conf4* consumed the lowest energy on average, since this configuration provided much faster execution, enabling power gating to provide significant energy savings due to the leakage power reduction. Therefore, these results show that an application's duty cycle should be considered when selecting configurations for execution. Larger configurations (e.g., *conf3* or *conf4*) that significantly reduce the duty cycle compared to smaller configurations provide greater power optimization benefits. However, when only short executions are required and an application's duty cycle is similar across the different configurations, and/or power optimization is not available (i.e., leakage power is high), the smaller configuration devices would consume less energy overall, since these configurations would typically have less leakage power than the larger configurations.

CONCLUSIONS AND FUTURE RESEARCH

The Internet of Things (IoT) is expected to grow at a fast pace, resulting in billions of connected devices that generate massive amounts of data. Due to this data explosion, the traditional IoT model, which involves edge nodes gathering and transmitting data to head nodes, will result in a communication bandwidth bottleneck, and latency and energy overheads. To ameliorate this overhead, edge computing performs computations on the edge nodes to interpret and utilize data, in order to reduce data transmission requirements, thereby reducing latency and energy consumption.

In this work, we seek to understand the microarchitectural characteristics that will support edge computing in the IoT. In order to understand the architectures, we must first understand the applications that will execute on these architectures. To tractably represent the vast IoT application space, we propose an application classification methodology consisting of a set of application functions and benchmarks that represent the basic computational patterns of current and emerging IoT applications. We comprehensively studied current low-power devices' microarchitectural characteristics and evaluated these devices' applicability to IoT edge computing. We evaluated these microarchitectural characteristics with respect to various performance metrics, and based on our analysis, we formulated insights that serve as a foundation for further analysis and design of IoT microprocessors. Since edge computing in the IoT is a burgeoning area of research, the goal of this work is to provide a foundation for further research, through the insights gained, into understanding application requirements and architectures that support edge computing.

Our analysis showed that an application's working set size should be given priority consideration over the input data size when designing IoT microprocessors. Additionally, we showed that current IoT-targeted microprocessors are not sufficient for edge computing, especially due to these devices' low memory capabilities (cache and main memory). In order to support edge computing, emerging IoT devices must be equipped with additional compute and memory capabilities. We also illustrated the need to prioritize the clock frequency and program execution order when designing IoT microprocessors, due to these

characteristics' large impacts on performance and energy consumption.

We also showed the importance of considering the executing applications' duty cycles when selecting configurations for IoT microprocessors, especially in the presence of optimization mechanisms, such as power gating. Large configurations that may increase the dynamic power, but reduce the duty cycle are preferred in such instances. However, where leakage power is high (e.g., where power gating is not available), smaller configurations that reduce the dynamic power are preferred. Thus, while designing high-performance embedded microprocessors to support edge computing, emphasis must also be placed on microarchitectural optimizations that reduce leakage power in order to realize the full benefits of these high-performance embedded systems.

Key next steps involve validating the analysis presented in this work in actual real-world use-cases. For example, we would like to study a surveillance camera use-case, where real-time face recognition applications must be implemented, and evaluate how these applications can be supported using the methodology presented in this paper.

Additionally, our work revealed some caveats that we intend to address in future work. For example, we plan to quantify the attainable performance benefits afforded by multicore architectures in the context of IoT edge computing. Furthermore, since we only considered the processing component in this study, we plan to extend the study to other IoT device components, such as input/output (I/O) bandwidth, secondary storage, etc., and evaluate how these components impact IoT edge computing. We intend to explore the tradeoffs involved in designing an architecture that provides runtime variability, such that the device can be dynamically configured to support various application functions, while minimizing the energy consumption. We also plan to study the impact of additional optimizations for low-power devices, and propose and prototype new architecture designs for IoT edge computing based on our analysis.

ACKNOWLEDGMENTS

The authors would like to thank Cullen Bash, Amip Shah, Martin Arlitt, Paolo Faraboschi, Dejan Milojicic, and Kevin Lim for their constructive feedback and assistance.

This work was supported by the National Science Foundation (CNS-0953447). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] I. Akyildiz, T. Melodia, and K. Chowdhury, "A survey on wireless multimedia sensor networks," IEEE Wireless Communications, vol. 14, Issue 6, December 2007.

[2] Asanovic et al., "The landscape of parallel computing research: a view from Berkeley," Technical Report No. UCB/EECS-2006-183, University of California, Berkeley, December 2006.

[3] Atmel Corporation. www.atmel.com

[4] Atzori L., Iera A., and Morabito G., "The internet of things: a survey," International Journal of Computer and Telecommunications Networking, October 2010.

[5] C.Bash, C. Patel, and R. Sharma, "Dynamic thermal management of air cooled data centers," Thermal and Thermomechanical Phenomena in Electronics Systems, May 2006.

[6] N. Binkert et al., "The gem5 simulator," Computer Architecure News, May 2011.

[7] Bonomi F., Milito R., Zhu J., and Addepalli S., "Fog computing and its role in the internet of things," Workshop on Mobile Cloud Computing, 2012.

[8] Cortex-M4 Processor – ARM. http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php

[9] Freescale Semiconductors. www.freescale.com

[10] Gartner Newsroom, http://www.gartner.com/newsroom/id/2684616

[11] E. Gaura, J. Brusey, and M. Allen, "Edge mining the internet of things," IEEE Sensors Journal, Vol 13, No. 10, October 2013.

[12] D. Giusto, A. Iera, G. Morabito, L. Atzori, "The internet of things," Springer, 2010.

[13] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (IoT): A vision, architectural elements, and future directions," Future Generation Computer Systems 29, pp. 1645-1660, 2013

[14] J. C. Haartsen, "The Bluetooth radio system," IEEE Personal Communications, vol 7, issue 1, pp. 28 – 36, February 2000.

[15] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, "Microarchitectural techniques for power gating of execution units," International Symposium on Low Power Electronics and Design (ISLPED), 2004.

[16] Intel Quark Technology. http://www.intel.com/content/www/us/en/processors/quark/intel-quark-technologies.html

[17] T. T.-O. Kwok and T.-K. Kwok, "Computation and energy efficient image processing in wireless sensor networks based on reconfigurable computing," Proc. Of the International Conference on Parallel Processing Workshops (ICPPW), Columbus, Ohio, August 2006.

[18] H. Lee et al., "Software defined radio – a high performance embedded challenge," High Performance Embedded Architectures and Compilers, Springer, 2005.

[19] W. Lehr and L. McKnight, "Wireless internet access: 3G vs. WiFi," Elsevier Telecommunications Policy, vol. 27, issues 5 – 6, pp. 351 – 370, July 2003.

[20] S. Li et al., "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," International Symposium on Microarchitecture, 2009.

[21] Y. Liu and S. K. Das, "Information-intensive wireless sensor networks: potential and challenges," IEEE Communications Magazine, vol. 44, no. 11, pp. 142-147, November 2006.

[22] T. Lovely, et al., "A framework to analyze processor architectures for next-generation on-board space computing," IEEE Aerospace Conference, March 2014.

[23] McKinsey Global Institute, "Disruptive technologies: advances that will transform life, business, and the global economy," www.mckinsey.com [Retrieved: July 2014]

[24] E. Nakamura, A. Loureiro, and A. Frery, "Information fusion for wireless sensor networks: methods, models, and classifications," ACM Computing Surveys, Vol. 39, Issue 3, 2007.

[25] C. Patel, C. Bash, R. Sharma, M. Beitelmal, and R. Friedrich, "Smart cooling of data centers," International Electronic Packaging Technical Conference and Exhibition, July 2003.

[26] V. Raghunathan, S. Ganeriwal, M. Srivastava, and C. Schurgers, "Energy efficient wireless packet scheduling and fair queuing," ACM Transactions on Embedded Computing Systems, February 2004.

[27] E. Sanchez-Sinencio, "Smart nodes of Internet of Things (IoT): a hardware perspective view and implementation," ACM Great Lakes Symposium on VLSI (GLVLSI), May 2014.

[28] S. Singh and R. Maini, "Comparison of data encryption algorithms," International Journal of Computer Science and Communication, vol. 2, no. 1, pp. 125-127, January 2011.

[29] STMicroelectronics. www.st.com

[30] S. Tilak, N. Abu-Ghazaleh, and W. Heinzelman, "A taxonomy of wireless microsensor network models," ACM Mobile Computing and Communications Review 6, pp. 28-36, 2002.

[31] M. Tory and T. Moller, "Rethinking visualization: a high-level taxonomy," IEEE Symposium on Information Visualization, 2004.

