Bolt: A Stateful Processor Interconnect

Felix Sutton, Marco Zimmerling, Reto Da Forno, Roman Lim, Tonio Gsell,
Georgia Giannopoulou, Federico Ferrari, Jan Beutel, and Lothar Thiele
Computer Engineering and Networks Laboratory
ETH Zurich, Switzerland
{firstname.lastname}@tik.ee.ethz.ch

ABSTRACT

The wireless sensor network community is currently undergoing a platform paradigm shift, moving away from classical single-processor motes toward heterogeneous multi-processor architectures. These emerging platforms promise efficient concurrent processing with energy-proportional system performance. The use of shared interconnects and shared memory for inter-processor communication, however, causes interference in the time, power, and clock domains, which prevents designers from fully harnessing these benefits. We thus designed Bolt, the first ultra-low-power processor interconnect for the compositional construction of heterogeneous wireless embedded platforms. This paper presents the architectural blueprint for interconnecting two independent processors, while enabling asynchronous inter-processor communication with predictable run-time behavior. We detail a prototype implementation of Bolt, and apply formal methods to analytically derive bounds on the execution time of its message passing operations. Experiments with a custom-built dual-processor platform show that our Bolt prototype incurs a negligible power overhead relative to state-of-the-art platforms, offers predictable message passing with empirical bounds that match the analytical ones to within a few clock cycles, and achieves a high throughput of up to 3.3 Mbps.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Multiprocessors—*interconnection architectures*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based System—*real-time and embedded systems*

General Terms

Design, Experimentation, Performance

Keywords

Cyber-physical systems; multi-processor; processor interconnect; predictability; composability; resource interference

1. INTRODUCTION

In the early days of sensor networks, platforms featuring a single 8- or 16-bit microcontroller (MCU), such as Mica [25] and Telos [43], spawned the development of a wide range of *sense-and-send* applications. These platforms offered modest computing resources well matched to the demands of low-rate sensing of temperature, humidity, light, etc. Low-power operation was achieved by interleaving sensing, data processing, and communication tasks and by judiciously managing the power state of hardware components.

Challenges. Although this design approach has been extensively followed to demonstrate the feasibility of wireless sensing applications, through our own experiences in designing, developing, and maintaining large-scale sensor network installations we have encountered recurring patterns that impede the construction and reliability of wireless embedded systems. We have observed that the engineering effort in realizing such systems is labor-intensive with respect to the design, test, and diagnosis of hardware and software components. While difficult to quantify, we argue that these practical complexities lead to implementations that are often unreliable, not readily adaptable to changing requirements, exhibit long development cycles, and are over-dimensioned to satisfy performance targets. This leads us to pose the following question: *Why is it difficult to design such systems?*

A careful analysis of existing wireless device architectures, in conjunction with a survey of the state of the art in the embedded systems design literature, reveals that the main problem is rooted in the *interference of hardware and software components in the time, power, and clock domains.*

One may think that simple sense-and-send applications do not suffer from this problem. However, also in these scenarios several tasks must be executed concurrently: reading sensors, processing data, transmitting packets, etc. *Sense-and-react* applications (*e.g.*, in cyber-physical systems [49]) additionally feature control and actuation tasks. These concurrent tasks interfere when they compete for shared resources such as clock cycles, memory, and peripherals. While labor-intensive engineering may partially handle such resource interference in highly deterministic scenarios, this approach is not only unsustainable in the long term but also ineffective if tasks are triggered by unpredictable events (*e.g.*, in surveillance or tracking scenarios [3]). Further, as the system load increases, the effects of resource interference increasingly affect the timing behavior of individual tasks, which in turn adversely impacts overall system performance. When incorporating power management techniques additional complexity is added to the system, further exacerbating the problem.

Contribution. To address the above challenges, we advocate a disruptive approach to the construction of future wireless embedded platforms. We propose the functional separation of tasks onto a multi-processor architecture whereby the tasks interact through asynchronous message passing using a processor interconnect with predictable timing characteristics. *Predictability* of the interconnect entails that passing a message takes a known, bounded time irrespective of the attached processors. As a result, the proposed architecture decouples processors in the time, power, and clock domains, facilitating the composable construction of customized platforms. *Composability* not only gives the system designer the flexibility to select hardware and software components satisfying the needs of the application, but also ensures that their interconnection does not change the properties of the integrated parts [27]. Furthermore, predictability of the message passing interface is essential to meeting the performance requirements imposed by certain application domains or wireless standards with tight latency constraints.

This paper presents BOLT, the first processor interconnect that enables the composable construction of ultra-low-power wireless embedded systems. BOLT provides predictable asynchronous communication between two arbitrary processors, and thus decouples the processors in the time, power, and clock domains. BOLT sits between both processors and hides all complexities associated with handling asynchronous message transfers from system developers. Two message queues, one for each direction, with first-in-first-out (FIFO) semantics form the core of this stateful interconnect. A signaling protocol allows for concurrent message reads and writes on both queues, and indicates when there is at least one message ready to be read out from a queue and when a queue is empty. The two message queues as well as all internal state reside in non-volatile memory, thus preserving the state of BOLT independent of the power states of the two processors. BOLT requires only a minimal software interface with well-defined semantics and predictable timing to be implemented on either processor. This concept of a stateful interconnect allows designers to choose arbitrary off-the-shelf or custom processors and "bolt" them together to create a customized dual-processor platform while avoiding resource interference.

We make the following contributions in this paper:

- Sec. 3 presents the design of BOLT.
- Sec. 4 validates our design by detailing a prototype implementation of BOLT on a state-of-the-art low-power MCU.
- Sec. 5 uses formal methods to derive bounds on the worst-case execution time of BOLT's message passing operations.
- Sec. 6 details BOLT's interface and how it should be used.

Using our BOLT prototype, we have built a heterogeneous dual-processor platform consisting of a 32-bit ARM Cortex-M4 and a 16-bit TI system on chip (SoC). Extensive experiments with this platform in Sec. 7 demonstrate the following: *(i)* BOLT achieves power decoupling while incurring negligible power overhead with a current drain of 430 nA during periods of inactivity; *(ii)* the execution times of message passing operations exhibit empirical bounds that match the analytical bounds to within a few clock cycles; *(iii)* BOLT provides a high throughput of 1.5–3.3 Mbps for inter-processor messages that are 16–128 bytes in length; and *(iv)* BOLT enables the smooth concurrent execution of event-triggered sensing and wireless communication—a scenario that is complex, labor-intensive, and error-prone to implement on current platforms. BOLT is available at `http://bolt.ethz.ch`.

2. BACKGROUND AND RELATED WORK

This section discusses the needs of low-power wireless embedded applications, the corresponding requirements on software components and hardware platforms, and the problem of resource interference in state-of-the-art architectures.

2.1 Classical Mote Architectures

In the early days of sensor networks, there was substantial interest in new hardware and software architectures for sensor nodes. These so-called *motes* typically provide the interface to sensors, and process and communicate data. Different hardware platforms have been developed both in industry and academia, exploring different points of the design space to match the demands of certain application domains (see, for example, Cricket [45], Mica [25], Telos [43], Iris [12], EYES [26], SunSPOT 9 [42], BTnode rev.3 [9], and TinyNode [13]). Attempts towards adaptivity at run-time often leverage multiple resources of the same type with different properties, for example, multi-radio platforms like BTnode rev.3 [9] or Opal [30] and the concept of wake-up radio [47]. Both the architectures and the design principles focused on finding the right trade-off between energy efficiency and the application-driven computing and communication requirements, interfaces to sensors, integration of state-of-the-art radios, small form factors, and autonomous operation [29].

2.2 Changing Requirements

Over the years the field of distributed low-power wireless embedded systems has matured to a point where now serious applications of societal and economic importance are within reach, such as the Internet of Things (IoT), industrial process control and supervision, environmental and structural monitoring, smart logistics, personalized medicine, home automation, and traffic control. In many of these applications, measurements are precious and must not be lost [52], so data must arrive reliably and in real-time [23, 46], responses are safety-critical [17], sensors are relatively expensive, and deployment and maintenance of a network is labor-intensive and costly. Thus, it is inevitable that distributed low-power wireless embedded systems become a high-quality infrastructure with known and predictable properties.

The requirements for successful hard- and software architectures in these domains are also much better understood than in the early days of SmartDust [31]. Developing software for heavily resource constrained hardware platforms is known to be highly labor-intensive due to the tight coupling between functional and non-functional properties on the one hand and detailed hardware properties on the other. Despite advances in model-based design techniques, specific models of computation, intensive distributed testing and verification cycles, distributed low-power systems are still error-prone. To make matters worse, the final installations are often embedded into hostile environments that are unknown at design time and exhibit dynamically changing properties.

The application domains differ substantially in their requirements and hence a single platform does not suite them all with respect to computational, communication, and memory resources, available energy budget, and degree of integration [5]. Nevertheless, we can identify two common trends.

- *Increasing resource demand.* The program and data memory required to implement even seemingly simple functionality has grown significantly. Similar observations hold for the required computational resources that very often can-not be fulfilled by the simple 8- or 16-bit MCUs that were employed in the first generation of motes.

- *Need for adaptability.* Application tasks such as communication, sensing, actuation, and computation are highly dependent on the occurrence of events (*e.g.*, on sensor and radio interfaces) and the availability of energy. The same holds for the associated modes of operation such as radio states, MCU power modes, duty cycle, and clock speeds.

One of the early attempts to increase the amount of available resources was the Imote2 [39], at the cost of high sleep currents limiting its use in energy-constrained settings. More recently, there are new MCU generations appearing on the market that combine low sleep currents, short wake-up times, and rich peripherals with a relatively high compute power, such as the ARM Cortex-M family [6] used in products from NXP, STMicroelectronics, Silabs, Freescale, and Atmel. But their appearance only solves part of the problem inherent to classical mote architectures: highly adaptive and event-triggered application tasks with widely varying resource requirements interfere on shared resources, thus violating the principles of modularity and separation of concern.

To illustrate this *resource interference* problem, let us consider an application that requires high-rate sampling of sensors, for example, acoustic sensors in a structural monitoring application [52]. Due to the fact that the single processing resource, the MCU, also needs to handle time-critical events from the wireless communication component, the computational power may not be sufficient and interference between tasks is inevitable. As a second example, suppose an application requires a node to react timely to events, for example, to quickly perform an actuation or to localize signal sources depending on the pairwise differences in the arrival of events at different nodes. Again, limited resources and interference on the single computing resource lead to unpredictable behavior in such adaptive scenarios. This kind of resource interference can only partly be hidden by software abstraction layers (*e.g.*, provided by TinyOS [37] and Contiki [14]), often leading to fragile, monolithic, and over-provisioned systems that are tedious to design, implement, debug, and maintain.

As a result, already early on in the history of sensor networks, modular architectures have been proposed that allow designers to seamlessly compose a node architecture from components that match the target application domain.

2.3 Modular Multi-processor Architectures

Early solutions to remove some of the deficiencies as described above have been the construction of modular architectures that can be adapted to the need of various application domains, including the 4-layer modular architecture in [44], the MIT Media Lab modular platform [8], the stackable architecture [41], and Epic [15]. In all of these designs, components communicate via a shared bus or a set of shared standard interconnects such as UART, I2C, SPI, and 1-wire. New shared interconnects like M-bus [35] have been specifically designed for ultra-low-power connectivity.

The need for heterogeneous systems consisting of multiple, possibly heterogeneous components that dynamically match resources to the specific task at run-time (*e.g.*, sleep modes, clock options, voltage scaling, component shut down) to allow for highly adaptive applications has been recognized by other communities as well, especially in the area of mobile communications. As a consequence, we increasingly see designs combining heterogeneous resources for ultra-low-power applications, such as the LPC4300 [40] from NXP, and the VF3xxR and MKW2xDx [21] from Freescale. All these designs use bus-based interconnects or shared-memory communication between the components. One notable exception is the TI F28M3x series [50], which is expressly designed for application domains requiring safety certifications. Instead of relying exclusively on shared resources for communication, it contains a FIFO buffer for conflict-free communication at the cost of not being aimed for ultra-low-power operation.

It has been recognized that whenever multiple resources communicate, the use of shared busses and shared memory seriously hampers modularity [32]. The major obstacle for application domains with high dependability and safety requirements, such as automotive and avionics, to adopt multicore and multi-resource platforms is the inevitable interference on shared resources [34]. Thus, there is currently no accepted path to certification, which requires guarantees on correct timing and function [51].

The main consequences of using shared memory or busses for communication among components are the following:

- *Coupling of power and clock domains.* To reach the goal of energy-proportional performance, where the energy consumed grows with the amount of useful task execution performed, systems use different power and clock domains. However, a shared bus couples these domains and requires tight coordination of power and clock management. As a result, the principles of separation of concerns and isolation of independent functionalities are violated, and power management requires interaction with many hardware and software layers [5]. As a rule, unnecessary interference and dependencies introduced on the hardware layer can rarely be decoupled by means of higher-level software constructs.
- *Interference in the time domain.* Using shared memory, constructs like semaphores and locks allow for mutually exclusive access to shared resources. However, they also make the timing of activities on one resource dependent on that of activities on another, seriously violating composability and the possibility of independent design. In other words, engineering must be re-done system-wide for each newly designed task. Similarly, the bus as a shared communication medium causes timing interference, leading to highly pessimistic timing bounds (*e.g.*, when using protocols like first-come-first-serve, fixed priority, round-robin) or inefficient communication (*e.g.*, when using partitioned protocols like TDMA, time-triggered architecture [33]).

Due to these deficiencies, alternatives like distributed memory, asynchronous message passing, and queue-based communication are all well understood in their underlying concepts and widely used in distributed systems, multi-processor systems, and networks on chip (NoC)[28, 22]. However, they have not yet been thoroughly investigated in ultra-low-power wireless embedded systems. This paper aims to fill this gap.

3. BOLT DESIGN

We introduce Bolt, a new processor interconnect for ultra-low-power wireless embedded systems. By providing bidirectional asynchronous message passing with predictable message transfer times, Bolt is a key building block for the communication between components on emerging dual-processor platforms to support ultra-low-power applications with high dependability requirements and stringent timing constraints. Bolt simplifies the design of such applications by completely eliminating or at least limiting the interference among differ-

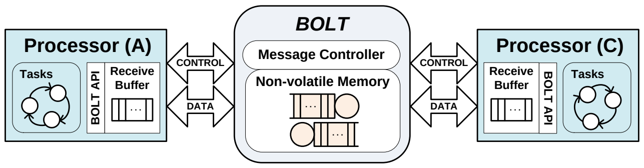

Figure 1: Overview of Bolt processor interconnect. By providing predictable bidirectional asynchronous message passing between processors A and C, Bolt decouples them in time, power, and clock domains.

ent hardware and software components on shared resources. As such, Bolt promotes a radical paradigm shift toward the *composable* construction and *predictable* operation of heterogeneous ultra-low-power wireless embedded platforms.

3.1 Overview

As depicted in Fig. 1, Bolt is a piece of integrated hardware and software that sits between two processors A and C. Bolt lets A and C asynchronously exchange messages while executing within their own time, power, and clock domains. This entails in particular that each processor can independently write messages into Bolt or read messages out of Bolt. We design and implement Bolt in such a way that the execution time of these read and write operations can be tightly bounded. Based on these bounds, we conceive a well-defined software interface by which each processor accesses Bolt in a non-blocking manner to exchange messages of variable length and possibly with different priorities.

The unique properties of Bolt allow for composable system designs. That is, a designer can choose any two commercially available processors and existing software artifacts and integrate them to create a customized platform that satisfies the application needs, *without changing the properties of the integrated parts.* These parts can be separately designed, implemented, and validated, thus leading to modular systems that are easier to develop, understand, and maintain.

Key to achieving these beneficial properties is to tackle the problem of interference between different hardware and software components on shared resources. To solve this problem, we take a disruptive approach in Bolt that is guided by the following established embedded systems design principles [24, 48]: *(i)* try to avoid interference; *(ii)* if interference is unavoidable, try to tightly bound it; and *(iii)* specify an interface with formally verified and predictable properties.

We describe next how to apply principles *(i)* and *(ii)* to the design of Bolt. Since principle *(iii)* depends on a concrete implementation of Bolt (see Sec. 4) and a formal analysis of its timing properties (see Sec. 5), we discuss the interface by which the processors access Bolt in Sec. 6. While the basic concept can be extended to more than two processors, we stick to the dual-processor case for illustration purposes and its immediate applicability to the separation of concurrent application (A) and communication (C) tasks.

3.2 Architecture

Bolt adopts asynchronous message passing to avoid interference between processors A and C wherever possible, as per design principle *(i)*. Thus, Bolt decouples processors A and C in the time, power, and clock domains.

- *Time.* Processors A and C interact with Bolt over dedicated *control* and *data channels*, as illustrated in Fig. 1. Using these channels, messages can be asynchronously transferred between each processor and Bolt, irrespective of the state of the other processor. Due to this and by buffering messages until they are read out, Bolt effectively decouples A and C in time.
- *Power.* Bolt stores undelivered messages and all internal state in non-volatile memory. Thus, it also decouples the two processors in power, enabling independent power management of A and C over a maximum dynamic range, including deep sleep and power off modes. Bolt can also retain its own state and undelivered messages after a complete power failure, which may suddenly occur, for example, in energy harvesting settings.
- *Clock.* Finally, there exist implementation choices for the data channel that allow A and C to select and adjust their clock frequencies, decoupling them in the clock domain.

To achieve these properties, the control and data channel between Bolt and the attached processors must be serviced by independent hardware blocks, thus enabling simultaneous message requests and message transfers from either processor. To support independent bidirectional message transfers, Bolt stores undelivered messages in two *FIFO queues*, as illustrated in Fig. 1. One queue is for messages written into Bolt by A and read out by C, while the other queue is for messages written by C and read by A.

Bolt's *message controller* manages the state of the message queues and the operation of the control and data channels. The message controller consists of software and hardware components. The latter can be realized on a multitude of hardware options, ranging from general-purpose MCUs to field-programmable gate arrays (FPGAs). A concrete message transfer is initiated asynchronously by one of the processors over the control channel, prior to the actual message transfer over the data channel.

Control channel. Operating the control channel means coordinating data channel access for message transfers and indicating the availability of messages to the target processor. The signaling sequence of the control channel, as described in Sec. 4, ensures that each processor can initiate message transfers without causing inter-processor interference in the power and clock domains. Indeed, each processor is free to trigger a read or write operation at any moment in time over the control channel, and transfer the message over the data channel without interfering with the concurrent execution of the other processor.

For example, at the end of a write operation, Bolt uses the control channel to inform the target processor that there is now a pending message. It is then up to the discretion of the target processor when to inspect the control channel and when to initiate a read operation. Whenever the target processor is not busy performing local operations, it can read the message from Bolt using its dedicated control and data channels irrespective of the state of the other processor.

By the same token, the power management of one processor is independent of the messaging operations invoked by the other processor. That is, one processor may, for example, choose to reside in a deep sleep mode for a given period of time as determined by its duty cycle, and is not explicitly woken up by the arrival of a message buffered in Bolt.

Data channel. The actual message transfer between a processor and Bolt occurs over a bidirectional data bus that supports master/slave operation. Two popular examples of standardized busses supporting master/slave operation are SPI and I2C. Each interconnected processor is the master

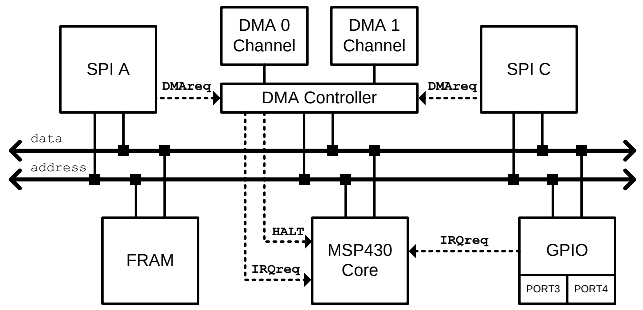

Figure 2: Simplified hardware block diagram of the MCU used in our prototype implementation of Bolt.

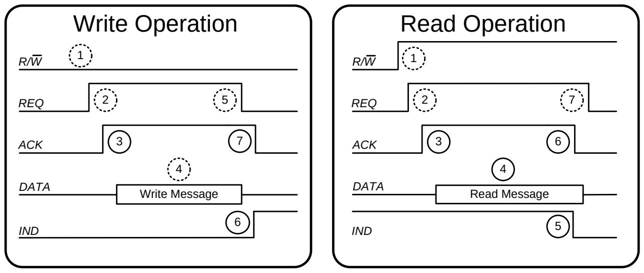

Figure 3: Signal sequences for write and read operations. The signals numbered with solid circles are driven by Bolt, while those numbered with dashed circles are driven by the interconnected processor.

of its dedicated data channel; that is, it provides the clock required to transfer each bit over the bus, while Bolt is always the slave. This ensures that each processor can transfer messages using its own independent clock frequency, thereby decoupling both processors in the clock domain.

Bounding unavoidable interference. Our architectural design decisions described thus far effectively avoid any interference between processors A and C in the power and clock domains. In the time domain, however, A and C may interfere. Since message transfers are asynchronous, it is possible that A and C simultaneously request a message operation on the *same* queue: one processor wants to read a message, while the other processor wants to write a message. This situation creates unavoidable interference at the message controller, because two processors compete for access to the same message queue (*i.e.*, a shared resource).

Following design principle *(ii)*, we intend to tightly bound the execution time of read and write operations despite this kind of resource interference. To this end, our design and implementation of Bolt strives to reduce as much as possible the hardware and software complexity of the message controller. By consequently following this guideline, we are able to accurately model the message controller and to determine tight bounds on the execution time of read and write operations using a model checker, as described in Sec. 5.

We acknowledge that if a message queue is full, a processor will not be granted access to write a message into Bolt, thereby blocking the write operation. We address this problem through the concept of virtual queues and appropriate buffer dimensioning, as discussed in Sec. 6.

4. BOLT IMPLEMENTATION

To demonstrate the viability of our design, we implement a Bolt prototype using the 16-bit TI MSP430FR5969 MCU running at an 8 MHz clock frequency. This particular MCU is well-suited due to its ultra-low power dissipation in sleep mode, the availability of built-in non-volatile storage in the form of ferro-electric random access memory (FRAM), an abundance of built-in peripherals, and its commercial availability at low cost. We next detail the hardware architecture of the MSP430FR5969, followed by a description of Bolt's software state machine that executes on top of it.

4.1 Hardware Architecture

Fig. 2 depicts a simplified hardware block diagram of the MSP430FR5969 including its built-in peripherals. The MCU is based on a Von Neumann architecture, where the core accesses program and data memory using a shared bus. The core, the FRAM, and all peripherals are attached to a shared 16-bit data bus and a 20-bit address bus. All program code, message data structures, and run-time variables are stored in the non-volatile FRAM of size 64KB. The instruction processing of the core may be interrupted by either the general-purpose input/output (GPIO) module or the direct memory access (DMA) controller using auto-vectored interrupt processing, whereby each peripheral interrupt is processed according to fixed priorities. The priorities are defined in hardware such that the DMA controller has a higher interrupt priority than the GPIO module.

The GPIO module has two ports, namely PORT3 and PORT4, each of which supports several input/output (I/O) lines. An I/O line can be individually configured to initiate an interrupt on either a rising or a falling-edge. If an interrupt condition is detected on either port, the GPIO module will inform the core through an interrupt request *IRQreq*. In the case of simultaneous interrupt conditions, hardware prioritization ensures that the interrupt service routine (ISR) associated with PORT3 is executed before that of PORT4.

The MCU provides two independent SPI hardware modules, namely SPI A and SPI C. The byte-wise transfer between the SPI receive buffer and the FRAM is coordinated by dedicated DMA channels, that is, DMA0 for SPI A and DMA1 for SPI C. The DMA controller manages the bus arbitration and interrupt priority, with DMA0 having a higher priority than DMA1. When an SPI module has received or transmitted a byte, a hardware trigger is signaled to the DMA controller *DMAreq* to initiate a memory transfer (*e.g.*, to store the received byte to FRAM or to fetch the next byte to transmit from FRAM). Before the DMA controller can perform such memory transfers, it must seize control of the shared bus from the core. The DMA controller initiates this takeover by halting the core through a hardware-driven request *HALT*. The core is allowed to complete the current clock cycle before it is halted, thus relinquishing its control of the address and data bus. The byte transfer between SPI and FRAM takes precisely two clock cycles, after which the halt request is cleared by the DMA controller and the core resumes operation as bus master on the next clock cycle. Once a fixed number of bytes has been transferred, as determined by the DMA's software configuration, the DMA controller will inform the core using an interrupt request.

Control channel. A dedicated GPIO port is used to implement the control channel toward each interconnected processor. Specifically, PORT3 is assigned to processor A, while PORT4 is assigned to processor C. We adapt a four-phase handshake protocol [38] to provide coordinated access for reading or writing a message over the data channel, as il-

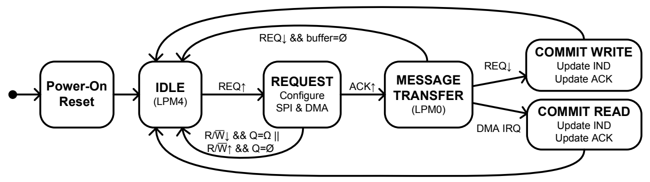

Figure 4: Bolt software state machine.

lustrated in Fig. 3. The control channel consists of the four lines $R/\overline{W}$, REQ, ACK, and IND. The $R/\overline{W}$ line defines the requested operation as either a message read or a message write. The REQ line is used by the interconnected processor to request the specified message operation, while the ACK line is used by Bolt to grant a message transfer over the corresponding data channel. If Bolt's message queues are empty or full, access to the data channel will not be granted for a read or a write operation, respectively. Bolt indicates to a processor when there is at least one message available to read using a dedicated IND line. The IND line is updated at the completion of every message operation.

Data channel. A dedicated SPI module is used to transfer messages between Bolt and each interconnected processor. Bolt configures both SPI modules in slave mode, thus decoupling each interconnected processor with respect to their clock domains. Dedicated DMA channels facilitate fast message transfers between each SPI module and the message data structures stored in FRAM.

4.2 Software State Machine

Fig. 4 shows Bolt's software state machine executing on the MSP430FR5969. Once the hardware is powered on, execution starts in a Power-On-Reset state where all state variables and message queues are initialized. If a queue contains undelivered messages, the IND line is asserted accordingly. The two REQ lines are configured as interrupt wake-up sources. The MCU is put into deep sleep (LPM4), while all other hardware blocks are turned off. The MCU remains in deep sleep until a read or write operation is initiated.

When either processor initiates a read or write operation by setting the $R/\overline{W}$ line and raising the REQ line, the associated IRQ wakes the MCU from deep sleep. Once the core is awake, the GPIO ISR will first determine the requested mode of operation and configure the corresponding SPI module and DMA channel for message transfer before updating the ACK line accordingly. The MCU will then enter a low-power mode (LPM0) with the core turned off, while the DMA and SPI modules remain active.

The completion of a message transfer is also processed by an ISR. However, depending on the message operation, either the GPIO or DMA ISR will be invoked. If the operation is a write, the falling edge of the REQ line signals the end of the message, thereby invoking the GPIO ISR. If the operation is a read, the DMA controller will trigger an interrupt toward the core. The ISR will update the internal data structures, update the ACK and IND lines, and disable the associated SPI and DMA peripherals. If there is an on-going message transfer involving the alternate processor, the MCU will return to the LPM0 sleep mode; otherwise it will return to the LPM4 deep sleep mode.

5. FORMAL TIMING ANALYSIS

As discussed in Sec. 3, two of Bolt's key design principles are to avoid resource interference wherever possible, and if interference is unavoidable, try to tightly bound it. Upper bounds on the interference on shared resources may be found using extensive measurements in a simulation environment or on an actual implementation of the system. However, this approach is extremely time-consuming and often infeasible in practice due to the number of possible combinations of input parameters and initial states of the system that need to be explored. An alternative approach, and one that has been well-studied in the embedded systems community, is to use formal methods such as model checking [7]. Through the construction of an accurate model of the system, one can apply rigorous mathematical tools to analytically derive safe bounds on interfering resources. We leverage such formal methods in verifying the timing predictability and functional correctness properties of Bolt.

In particular, we show next that Bolt has tightly bounded worst-case execution times for read and write operations, and that the physical interface operates according to its specification. We start by identifying the relevant hardware and software components of the Bolt prototype, and construct a model of their interactions using a network of timed automata [4]. We then parametrize the model based on our prototype and formally verify the timing and functional properties of Bolt using the Uppaal model checker [36].

5.1 Model Checking

The challenge in constructing a model of our Bolt prototype is capturing the complexity of several time-dependent and interacting state machines each of which has their own independent clock domain. In particular, processor A and C initiate message operations within their own clock domain, while the Bolt MCU responds to these operations using its own clock. Furthermore, contention between the execution of GPIO and DMA ISRs need to be formally modeled.

As a solution to this challenge we propose to model all interactions as timed automata and to extend each automaton with an independent clock variable. This technique is based on the theory developed in [4], where networks of time-dependent state machines interact through synchronization channels. This formalism allows us to model the complex interactions between Bolt and the interconnected processors by incrementing all clock variables synchronously.

We use Uppaal, a popular toolbox for the modeling, simulation, and verification of timed automata networks [36]. We construct a run-time model of our Bolt prototype and its interactions with two interconnected processors. The run-time model consists of four interacting timed automata:

- the processor automaton
- the Bolt software state machine automaton
- the GPIO port automaton
- the DMA channel automaton

In the following, we introduce the behavior modeled by each automaton. Due to space constraints, we limit detailed explanations to the Bolt software state machine automaton.

5.1.1 *Processor Automaton*

The processor automaton represents the expected signaling sequence of a read or write operation, according to the specification in Sec. 3. There are two instances of this automaton, one for each interconnected processor, each with its own clock variable. The type of operation (*i.e.*, read, write, or no-operation) and the time instant at which an operation is started are specified as non-deterministic transi-

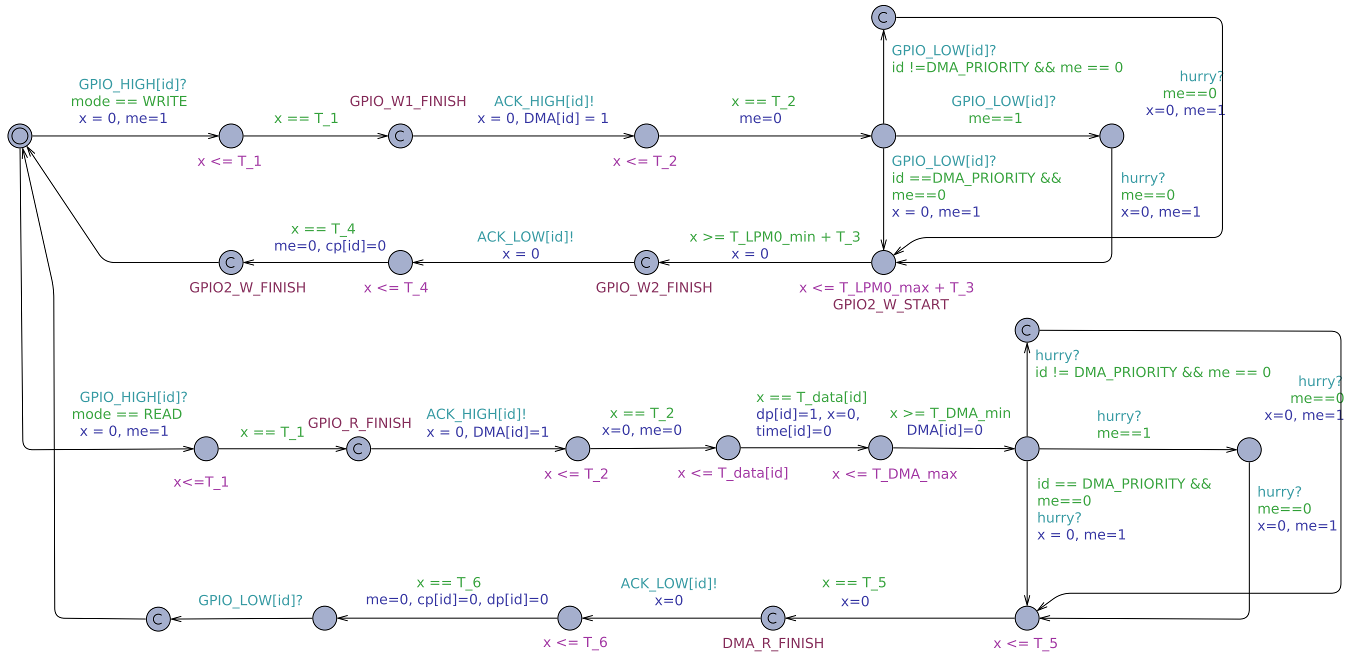

Figure 5: Timed automaton modeling the Bolt software state machine of Fig. 4 within the Uppaal toolbox.

tions. This level of non-determinism ensures that we indeed capture the worst-case timing operation of Bolt. Note that during verification, the Uppaal model checker will exhaustively explore all feasible scenarios and concurrent activities.

5.1.2 *Bolt Software State Machine Automaton*

The Bolt software state machine automaton represents the software execution of the Bolt message controller. The automaton, shown in Fig. 5, uses features like state invariants and transition guards provided by Uppaal [36] to model the execution time of each GPIO and DMA ISR.

Variable transition times, such as the wake-up time from the LPM0 low-power mode of the core, are incorporated into the automata by limiting the invariant to at most the maximum delay, while relaxing the guard to at least the minimum delay. In this way, the Uppaal tool is free to choose an arbitrary delay within the specified bounds and can explore all possible transitions. The execution of either GPIO and DMA ISRs is protected by a binary semaphore in the software state machine. This ensures that only one ISR can be executing at any moment in time. The prioritization of the ISR execution is defined by the GPIO port automaton.

5.1.3 *GPIO Port Automaton*

The purpose of the GPIO automaton is to model the hardware prioritization between the two GPIO ports. The final state machine model of Bolt contains two GPIO automaton instances, one for each processor port. They model the detailed protocol interactions and the case of simultaneous message requests by the two connected processors. The corresponding priority-based arbitration is taken into account as well as the fact that the GPIO port has lower hardware priority than the DMA controller; that is, if a DMA interrupt is pending, the GPIO port automaton will ensure the DMA ISR acquires the semaphore first. Another important behavior modeled by this automaton is the time to wake-up from LPM4. We use global state variables to ensure the wake-up time is bypassed when there are ongoing operations.

5.1.4 *DMA Channel Automaton*

Once a GPIO port has taken the binary semaphore, the execution time of the ISR may be extended due to a DMA memory transfer between the SPI peripheral bus and the FRAM. The DMA channel automaton ensures that the hardware prioritization between each DMA channel is adhered to. As described in Sec. 4.1, when there is a byte to transfer between the SPI and the FRAM, the DMA controller stalls the MCU for two clock cycles. We model this using Uppaal's stop-watch feature [11], whereby a timed automaton can stop the clock of another automaton. This feature enables each DMA channel to stop the clock associated with the Bolt automaton for precisely two clock cycles before allowing it to resume. In this way, we precisely capture the interactions described for the MSP430FR5969 in Sec. 4.

5.1.5 *Complete System Model*

Table 1 lists the number of states, transitions, and clock variables of the individual timed automata templates. The complete system model comprises of nine automaton instances: two processor automata (for processors A and C), two Bolt software state machines, two GPIO port automata, two DMA channel automata, and one supplementary automaton that enforces the urgent selection of specific transitions in the network of timed automata. The resulting model of the complete system consists of 125 states, 165 transitions, 8 clock variables, and 15 synchronization channels.

Table 1: Timed automata in the Bolt system model

Automaton	#States	#Transitions	#Clocks
Processor	14	13	1
Bolt software	24	28	1
GPIO port	19	32	1
DMA channel	5	9	1

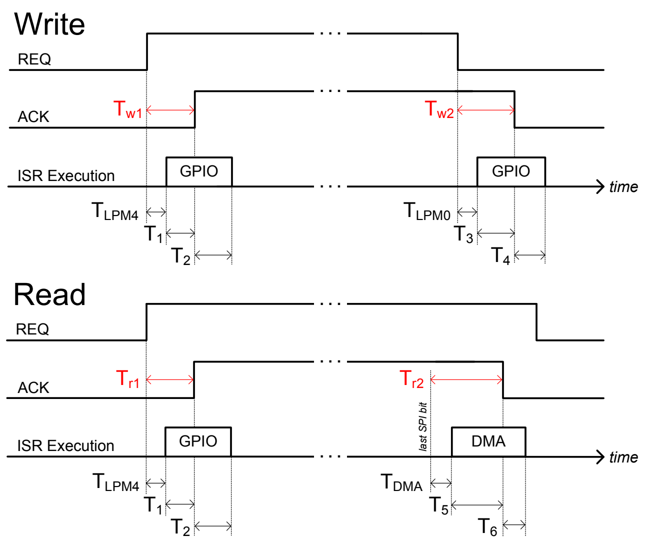

Figure 6: GPIO and DMA interrupt service routine execution with respect to read and write operations.

5.2 Timing Parameter Characterization

The timed automata templates model the high-level hardware/software operation of BOLT, parametrized using low-level timing parameters. We now describe the methodology we use to obtain safe bounds on these quantities.

Fig. 6 shows the execution of the GPIO and DMA ISRs with respect to the *REQ* and *ACK* lines for both read and write operations. To determine the specific worst-case delays in terms of clock cycles of the MSP430FR5969, we first need to investigate two important issues: *(i)* unbalanced ISR execution time, and *(ii)* non-deterministic hardware delays.

Since the GPIO and DMA ISRs must always determine which of the two processors is to be serviced, conditional statements are needed to select the desired control flow, resulting in the ISRs exhibiting port-specific execution times. By inserting the appropriate number of no-operation instructions into the ISR assembly code, we ensure that the GPIO and DMA ISRs execute a constant number of clock cycles, irrespective of which processor triggers the interrupt.

The next problem to address is the MCU's delay in waking up from the low-power sleep modes, denoted by T_{LPM4} and T_{LPM0}, and the time it takes from signaling a DMA interrupt until the beginning of an interrupt context, denoted by T_{DMA}. Typically, these hardware-specific delays are specified in the datasheet of the MCU as being deterministic. However, detailed measurements on our BOLT prototype with a logic analyzer show that the particular delays concerned are in fact non-deterministic within a small bounded range. Fig. 7 shows the histograms of the measured delays expressed in clock cycles of the MSP430FR5969 MCU.

With the measured upper and lower bounds of the timing delays, as listed and defined in Table 2, the formal system model is fully parameterized. The timed automata model can now be used to determine the worst-case timing bounds on the complex run-time dynamics of BOLT.

5.3 Timing Predictability Analysis

To find the worst-case execution time for read and write sequences, that is, T_{r1}, T_{r2}, T_{w1} and T_{w2} as annotated in Fig. 6 and defined in Table 2, we extend the Uppaal model in three ways. First, we add an additional global clock

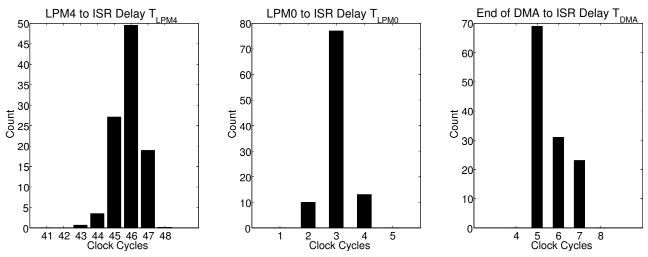

Figure 7: Histogram of wake-up delays T_{LPM4}, T_{LPM0}, and DMA interrupt delay T_{DMA} in MCU clock cycles.

Table 2: Measured timing parameters of the system model in MCU clock cycles as annotated in Fig. 6

Parameter	Description
$T_{LPM4} \in [41,48]$	Wake-up from LPM4 until beginning of ISR execution
$T_1 = 172$	Start of GPIO ISR until rising edge of the *ACK* line
$T_2 = 48$	Rising edge of the *ACK* line until the end of the GPIO ISR
$T_{LPM0} \in [2,4]$	Wake-up from LPM0 until beginning of ISR execution
$T_3 = 149$	Start of the GPIO ISR until falling edge of the *ACK* line
$T_4 = 58$	Falling edge of the *ACK* line until the end of the GPIO ISR
$T_5 = 117$	Start of the DMA ISR until the falling edge of the *ACK* line
$T_6 = 59$	Falling edge of the *ACK* line until the end of the DMA ISR
$T_{DMA} \in [5, 7]$	Last bit on the SPI bus until the beginning of the ISR execution

variable to support the construction of verification queries using a common time base. Second, we modify the processor automaton such that the Uppaal model checker can non-deterministically select the type of operation (*i.e.*, read, write, or no-operation) performed by each processor instance. Third, we introduce a delay at the beginning of each operation and allow the model checker to non-deterministically select the delay of each operation. By selecting an appropriate upper bound for the delay (*e.g.*, double the expected duration of a single operation), it is assured that all possible singular and simultaneous operations, with all possible relative timing offsets, are explored with clock cycle resolution.

We determine the worst-case execution times by submitting queries to the Uppaal model checker. Specifically, we query the existence of a global time value that exceeds a given threshold X for a specific BOLT instance and state. We apply binary search to find the minimum threshold X that satisfies the query. Table 3 shows the worst-case execution times for singular read and write as well as simultaneous read and write operations as derived from the Uppaal model.

We now use these values to determine the worst-case execution time for any BOLT operation, irrespective of the type of operation performed. We define the worst-case execution time of the request phase $T_{request}$ and the commit phase T_{commit} in terms of the maximum execution time of read and write sequences as defined in Fig. 6. The worst-case execution times are defined by the following expressions:

$$T_{request} = \max(T_{r1}, T_{w1})$$
$$T_{commit} = \max(T_{r2}, T_{w2})$$

Table 4 summarizes the results for each GPIO port instance. As expected, the request phase execution time dif-

Table 3: Analytically determined worst-case execution times for singular reads and writes and simultaneous read & write operations in MCU clock cycles

	Write		Read		Read & Write			
	T_{w1}	T_{w2}	T_{r1}	T_{r2}	T_{w1}	T_{w2}	T_{r1}	T_{r2}
PORT3	220	153	220	124	418	397	418	357
PORT4	220	153	220	124	466	397	466	357

Table 4: Worst-case execution times in MCU clock cycles for request and commit phases, irrespective of whether Bolt executes a read or a write operation

	$T_{request}$	T_{commit}
PORT3	418	397
PORT4	466	397

fers between the two GPIO ports due to different priorities. The execution time of the commit phase is identical for both GPIO ports, since the DMA controller halts the core for the same number of clock cycles under the worst-case scenario.

5.4 Functional Correctness

We define functional correctness of Bolt in terms of the asynchronous signaling sequence defined in Sec. 3. Given a valid input signal sequence, Bolt should always provide a correct output sequence. In terms of the developed timed automata model, the formal validity of Bolt's functional correctness is assessed based on the existence or absence of a deadlock. Here, a deadlock denotes a state in which no edge transition can be executed by any automaton in the system model, that is, the system "stalls." After verifying the absence of such a deadlock in the developed model by submitting an appropriate query to the Uppaal model checker, we can conclude that the model developed does not exhibit a state sequence leading to a deadlock.

6. BUILDING APPLICATIONS WITH BOLT

The stateful processor interconnect provided by Bolt is a stepping stone toward building highly reliable, resource-efficient, and timing-critical wireless embedded applications. Bolt concentrates on contributing a communication mechanism with well-defined semantics and known timing properties. We now describe the interfaces as well as the necessary implications for a successful integration with Bolt.

6.1 Interface Specification

The complexity of real-world wireless sensing applications implies that multiple message types, coupled with the possibility for prioritization between message types, may be required. For example, messages containing protocol data may be considered low priority, while messages containing protocol control information may be considered high priority.

In the case of only a single message priority, the problem of non-blocking writes required by our asynchronous message passing scheme can be solved by correctly dimensioning the available buffer space based on the expected read and write dynamics associated with the application domain.

If instead multiple message priorities are needed, the problem becomes significantly more complex. We thus introduce an abstraction called *virtual queues* to reduce the problem to the single priority-based message flow problem. We consider a virtual queuing system, as shown in Fig. 8. Messages of priority i are written into a queue of length B_i, and are

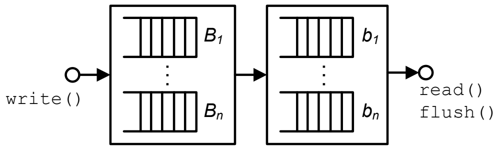

Figure 8: Message passing through virtual queues.

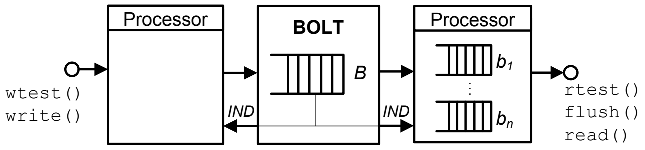

Figure 9: Realization of virtual queues using Bolt.

read from a separate queue of length b_i. We define the rate at which all messages in queue B_i are transferred into queue b_i as the *flush rate*. Under this construction, it is possible to compute the queue lengths B_i and b_i for each priority i under the assumption of a minimum flush rate.

Fig. 9 shows how the virtual queue abstraction is mapped onto a concrete message queue in Bolt. The length of the Bolt message queue B must be greater than or equal to the aggregate queue sizes of all prioritized queues, $B \geq \sum_{i=1}^{n} B_i$, while the receive buffer must be at least as large as all prioritized queues, that is, $b_i \geq B_i$ for all message priorities i.

Virtual queues make it possible for a single processor to handle more than one input queue, *e.g.*, in the case where Bolt is extended to more than two processors. With an appropriately dimensioned flush rate, the virtual queue abstraction coupled with an aggregate *IND* line, *i.e.*, representing the availability of a message in any of the pending message queues, Bolt facilitates non-blocking message operations. If more than one thread having data dependencies executes on a single processor, asynchronous batch processing [10] may be applied to guarantee timing constraints. However, if more than one thread performing Bolt message operations executes on a single processor, a non-preemptive round-robin scheduler may be employed to ensure non-blocking message operations with predictable timing behavior.

6.2 Application Programming Interface

Bolt provides a simple application programming interface (API) as shown in Table 5 (see also Fig. 9). Besides `read` and `write`, it includes the three additional functions `rtest`, `wtest` and `flush`. The following preconditions must be met by each interconnected processor when using Bolt.

- There must be space for at least one message in Bolt before `write` is called.
- There must be at least one message pending in the receive buffer before `read` is called.
- There must be no buffer overflow of the receive buffer.

The first precondition is satisfied by maintaining a local variable on each attached processor that stores the number of free messages in Bolt. This variable is initialized when the *IND* line is low, is decremented by one at each `write` invocation, and is accessed using the `wtest` function. It follows that `wtest` must return a positive integer before a `write` is invoked. The second precondition is satisfied by storing a counter for the number of messages available in the receive buffer for each message priority i. A `read` therefore can only be invoked if `rtest` returns a positive integer for a specified priority level. The last precondition is satisfied by

Table 5: Bolt application programming interface

Function	Description
`msg_t* read(int i)`	Read a message of priority i from receive buffer
`void write(msg_t *m, int i)`	Write a message of priority i into BOLT
`int rtest(int i)`	Return number of messages of priority i in receive buffer
`int wtest(int i)`	Return number of free spaces in BOLT available for messages of priority i
`void flush(void)`	Read all messages from BOLT into receive buffer

invoking `flush` at a rate no greater than that determined by a queue buffer size analysis.

If a message queue is empty upon a read request or the message queue is full upon a write request, the BOLT state machine will not raise the *ACK* line in response to the raising of the *REQ* line. Instead, the BOLT state machine will safely return to an idle state and wait for the next request, while the requesting processor may stall indefinitely waiting for the *ACK* line. Nevertheless, the BOLT API ensures that an interconnected processor can perform a non-blocking read or write operation. This is done by monitoring the state of the *IND* line, maintaining local message counters, and appropriately executing the `wtest`, `rtest` and `flush` functions.

6.3 Message Consistency

In a real-world system, the loss of power or the reset of a processor can lead to a loss of state. While the issue of reconstructing state or managing erroneous state (*e.g.*, after a reboot) is left to higher layers, BOLT ensures that the messages it maintains are consistent. Specifically, the architecture and interfaces of BOLT guarantee functional correctness of the processor interconnect in case of a spontaneous loss of power. This is achieved by storing both the BOLT message queues and all associated state in non-volatile memory, which will persist through a loss of power. Furthermore, the BOLT state machine ensures that partially read messages are not removed from the message queue, while partially written messages are removed from the message queue. This behavior ensures that undelivered messages can always be safely retrieved from BOLT when power is re-established.

7. EVALUATION

This section evaluates the performance of our BOLT prototype using extensive experiments with a custom-built dual-processor platform. First, we focus on a single BOLT instance and characterize its operation in terms of power decoupling, power overhead, timing predictability, and message throughput. Then, we demonstrate the benefits of BOLT in a typical event-triggered wireless sensing scenario.

7.1 Custom-built Dual-processor Platform

Fig. 10 shows a custom-built heterogeneous dual-processor platform, where BOLT interconnects two state-of-the-art processors with different computing capabilities and power dissipation: a 32-bit STM ARM Cortex-M4 STM32F303VCT6 running at 72 MHz and a 16-bit TI CC430F5137 SoC running at 20 MHz. We constructed this platform by connecting

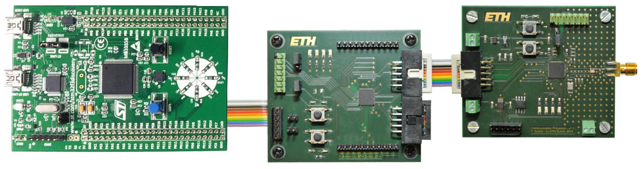

Figure 10: Custom-built heterogeneous dual-processor platform used in the experiments. The Bolt prototype interconnects a 32-bit ARM Cortex-M4 (left) with a 16-bit TI CC430 SoC (right).

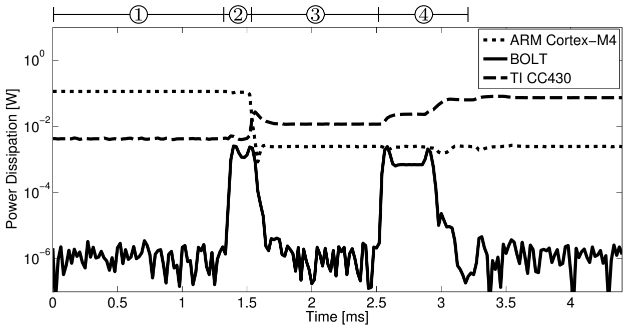

Figure 11: Power profiles of the ARM Cortex-M4, the TI CC430, and our Bolt prototype.

BOLT's control (4 I/O lines) and data (3-wire SPI bus) channels to each processor, and developing a small software module for each processor that implements the read and write operations as described in Sec. 4.1. We use this platform in the following experiments.

7.2 Power Decoupling

We start by illustrating how BOLT decouples two processors in the power domain. To this end, we perform an example execution with the dual-processor platform, using an Agilent N6705A DC power analyzer to measure the current of our BOLT prototype, the Cortex-M4, and the CC430 at a supply voltage of 3.0 V and a sampling rate of 48 kHz.

Execution. Fig. 11 shows the power dissipation of the three cores over a period of 4.5 ms. We can distinguish four phases.

In *phase* ①, BOLT and the CC430 reside in a low-power sleep mode, while the Cortex-M4 collects sensor measurements by periodically sampling its built-in analog-to-digital converter (ADC). BOLT dissipates approximately 1.3 μW.

In *phase* ②, once enough samples are collected, the Cortex-M4 writes a single message into BOLT using an SPI frequency of 4 MHz. When the Cortex-M4 initiates the write operation, BOLT goes from deep sleep into active mode, where it dissipates about 1.1 mW. At the end of the write operation, BOLT asserts the *IND* line to indicate to the CC430 that a message is pending and then returns to deep sleep.

At the beginning of *phase* ③, the CC430 wakes up, initializes its wireless transceiver, and waits until the communication channel is available. During this time, the Cortex-M4 and BOLT both reside in energy-saving sleep modes.

In *phase* ④, the CC430 reads the pending message from BOLT using a 2 MHz SPI clock and proceeds to transmit the contents of the message over its wireless radio.

Finding. The above execution shows that BOLT decouples two processors in the power domain. Indeed, each processor is free to locally decide when to enter or awake from a sleep mode, irrespective of the current power state and activity of the alternate processor and BOLT itself. This enables optimal power management over a maximum dynamic range.

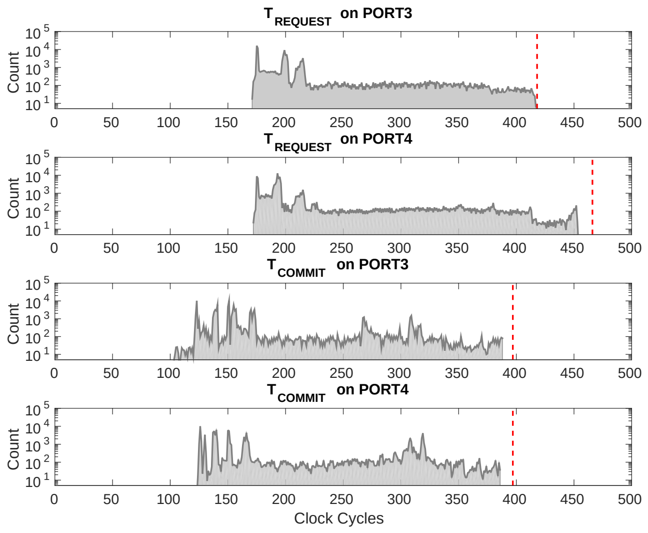

Figure 12: Histograms of the execution times of request and commit phases for concurrent message operations by two interconnected processors. Vertical dashed lines indicate the upper bounds from Table 4 determined with the Uppaal model checker.

7.3 Power Overhead

Using power measurements from the previous experiment, we quantify the power overhead of our Bolt prototype.

Results. When Bolt does not perform reads or writes, its power dissipation of 1.3 μW is less than state-of-the-art low-dropout voltage regulators, and, in our setup, several orders of magnitude lower than the sleep modes of both interconnected state-of-the-art processors, as visible in Fig. 11. Furthermore, Bolt's *active* power dissipation of 1.1 mW is comparable to the *sleep mode* power dissipation of the Cortex-M4. We thus conclude that Bolt incurs a negligible power overhead in a dual-processor platform design.

7.4 Timing Predictability

In a second experiment, we empirically verify the analytical bounds on the execution time of read and write operations determined by the Uppaal model checker in Sec. 5.

Setting. The time to complete a read or a write operation depends on the time to transfer the message over the SPI bus and the time Bolt needs to execute its interrupt-driven state machine according to the request and commit phases of the asynchronous interface, as detailed in Sec. 4. While the message transfer time is given by the message length and the SPI frequency, the duration of the request and commit phases, $T_{request}$ and T_{commit}, are non-deterministic due to interference on the same message queue in case of simultaneous accesses by both interconnected processors.

We accurately measure $T_{request}$ and T_{commit} on our Bolt prototype using a logic analyzer sampling at 25 MHz, while the Cortex-M4 and the CC430 perform concurrent message operations. We configure each processor to write a fixed number of messages into Bolt, before reading out all pending messages. The Cortex-M4 writes 48-byte messages and the CC430 writes 24-byte messages, resulting in equal message transfer times over the SPI busses. In total, 100,000 simultaneous message operations are performed, using a randomized waiting time between successive message operations to stress-test Bolt and to generate different access patterns.

Table 6: Maximum queue size and average throughput of Bolt prototype for different message lengths

Message length [bytes]	16	32	48	64	128
Max. messages per queue	1075	568	380	290	148
Avg. throughput [Mbps]	1.5	2.1	2.5	2.8	3.3

Results. Fig. 12 shows historgrams of $T_{request}$ and T_{commit} measured for the Cortex-M4 attached to PORT3 and the CC430 attached to PORT4. We see the analytical bounds are extremely tight: they are at most 12 clock cycles greater than the worst-case execution times we measured. This can be attributed to our quest for simplicity in the design and implementation of Bolt as well as our accurate modeling.

Further, in accordance with the model, the worst-case execution time of the request phase, $T_{request}$, is slightly longer on PORT4 than on PORT3 (48 MCU clock cycles), because of the interrupt prioritization of PORT3 over PORT4. Instead, the worst-case execution time of the commit phase, T_{commit}, is approximately equal for both ports. This is due to the fact that DMA memory transfers performed during the commit phase, as described in Sec. 5.3, halt the MCU for an equal number of clock cycles for each DMA channel.

In summary, the results demonstrate that our Bolt prototype exhibits timing-predictable read and write operations, satisfying a key requirement for enabling composability.

7.5 Message Throughput

Next, we quantify the maximum message throughput supported by our Bolt prototype.

Setting. We determine the throughput by measuring the time to write a sequence of messages into Bolt at a SPI frequency of 4 MHz. Specifically, we let the Cortex-M4 write 1000 messages into Bolt, and measure the message transfer time with a logic analyzer. We consider write operations as they take longer than read operations: a message can be read 29 MCU clock cycles faster than it can be written.

Results. Table 6 shows the maximum queue size (number of messages that fit into a queue) and the average throughput for different message lengths. We see that the throughput increases with the message length, supporting up to a maximum of 3.3 Mbps for 128-byte messages. Assuming the available memory in Bolt is evenly allocated to both FIFO message queues, the number of messages that can be stored into each queue is inversely proportional to the message size, allowing up to 1075 16-byte messages to be stored. Thus, by choosing a suitable message length, Bolt can support applications with high inter-processor communication demands.

7.6 Use Case: Event-triggered Sensing

In a final experiment, we demonstrate the benefits of Bolt in a typical event-triggered wireless sensing scenario.

Scenario. The application demands resource-constrained embedded devices to handle unpredictable *sensor events* originating from a physical process, while simultaneously servicing the wireless network interface to report those events over the radio. Upon the occurrence of an event, a device starts to acquire ADC samples to classify the event, and constructs a message containing all metadata associated with the event. The time between event detection and the first ADC sample should be as short as possible so as to maximize the number of samples that contain useful data about the event. To report events and coordinate their operation, devices run a low-power wireless communication protocol.

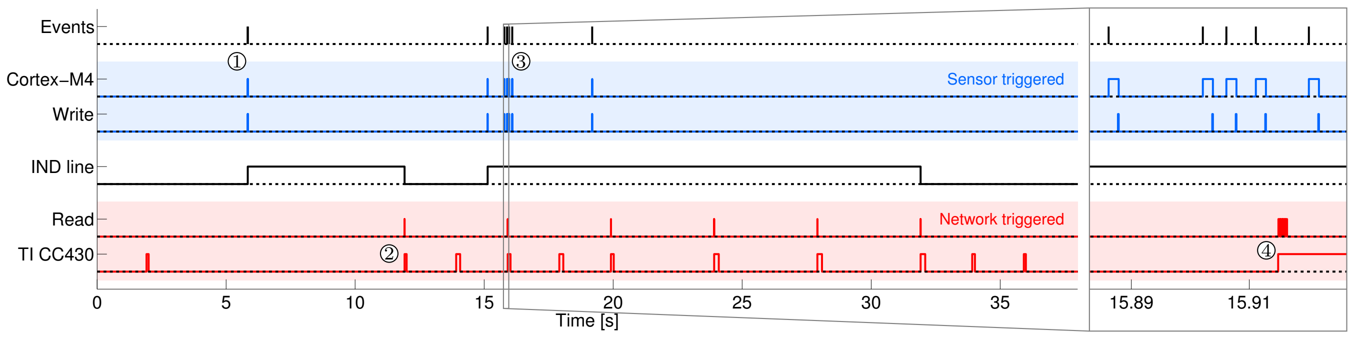

Figure 13: Signal trace extracted from an example event-triggered wireless sensing application using Bolt.

Given the timing constraints inherent to many protocols and standards, such as scheduled wake-ups [16, 18] and communication slots [1, 2], devices execute subject to hard real-time deadlines. During periods of inactivity, devices reside in a low-power sleep mode to meet the application's lifetime goal.

Without Bolt. Realizing this application based on a single-processor platform would lead to resource interferences whenever sensing and networking events must be handled at the same time, resulting in degraded performance or even erroneous behavior. By contrast, partitioning sensing and networking tasks onto two different processors improves a device's capability to handle both types of events in a timely manner. However, using a shared memory or a shared bus for inter-processor communication would entail a tight coupling of the two processors in the time, power, and clock domains, with all the drawbacks discussed in Sec. 2.

With Bolt. Using Bolt as the interconnect avoids these issues and provides maximum flexibility in selecting appropriate processors for each task, for example, the Cortex-M4 as a capable application processor and the CC430 as a ultra-low-power communication processor. Furthermore, software implementation and re-use are simplified, processors can independently switch power modes while exchanging messages within guaranteed timing bounds, and the overall system becomes significantly more robust and easier to maintain.

The costs of Bolt relative to directly connecting two processors via I/O lines and a SPI bus is application-specific and difficult to quantify. Besides the additional cost of the MCU, there is an increase in the overall hardware surface area; our Bolt prototype requires 49 mm^2 plus support circuitry. The total power dissipation of the platform increases by 1.3 μW during periods of no message exchange. Each message transfer consumes about 350 nJ assuming 128-byte messages at a 4 MHz SPI frequency, while the message throughput reduces by about 17 % compared to an optimally configured SPI bus and under the best-case assumption that both processors are always ready to service the SPI to read and write messages. In our view, the benefits of Bolt far outweight these costs.

Bolt in action. We implemented the above application on our custom-built dual-processor platform. The CC430 runs the Low-power Wireless Bus (LWB) [19], a communication protocol that uses fast and highly reliable Glossy floods [20] to transmit packets among nodes in a multi-hop network.

Fig. 13 shows a representative execution trace illustrating the interaction of the two processors through Bolt. The top three rows show the arrival of sensor events, the servicing of sensor events, and the writing of event messages into Bolt by the Cortex-M4. The fourth row indicates the state of Bolt's *IND* line towards the CC430. The last two rows show the reading of event messages from Bolt and the processing of network-triggered events by the CC430.

Looking at the first and the last row, we see that sensor events occur randomly and possibly in bursts, while network events occur periodically with the period changing according to the required communication bandwidth.

By inspecting the active phases of both processors, we see that Bolt allows them to act independently of each other. For example, at time ①, a sensor event triggers the Cortex-M4 to wake up from sleep mode and to write a new message into Bolt. The *IND* line indicates a pending message to the CC430. This message is then asynchronously requested from Bolt by the CC430 at time ②, where it starts to interact with the network as determined by LWB's communication schedule [19]. The independent handling of sensor and network events ensures task deadlines are adhered to, even in cases of heavy event bursts as visible at time ③ in Fig. 13.

As shown in Sec. 5.3, the time to perform message operations with Bolt can be tightly bounded. In this use case, this property enables the communication processor to read a sequence of pending messages just before they are sent over the radio via Glossy at time ④, thereby limiting the number of wake-up cycles performed by the communication processor, and thus reducing the platform's energy footprint.

8. CONCLUSIONS

In this paper, we focus on the resource interference problem in today's wireless embedded platforms. We argue that to address this problem, a multi-processor architecture built around an interconnect offering asynchronous message passing with guaranteed message transfer times is needed. Based on this idea, we design and implement Bolt, a stateful interconnect for ultra-low-power wireless embedded systems. We show how our Bolt prototype avoids interference in the power and clock domains, while exhibiting tight bounds on unavoidable interference in the time domain between two interconnected processors. Evaluation using a custom-built heterogeneous dual-processor platform shows that Bolt can fully decouple the interconnected processors while providing high-throughput, timing-predictable inter-processor communication at a negligible power overhead. We thus maintain that the architectural blueprint provided by Bolt is an important building block for the composable construction of next-generation wireless embedded platforms.

Acknowledgments. We thank Luca Mottola, the anonymous reviewers, and our shepherd Jie Liu for valuable feedback. This work was scientifically evaluated by the SNSF and financed by the Swiss Confederation and Nano-Tera.ch.

9. REFERENCES

[1] HART communication foundation. `http://en.hartcomm.org/main_article/wirelesshart.html`.

[2] IEEE 802.15.4e Wireless Standard - Amendment 1: MAC sublayer. `http://standards.ieee.org/findstds/standard/802.15.4e-2012.html`.

[3] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 1994.

[5] M. P. Andersen and D. E. Culler. System design trade-offs in a next-generation embedded wireless platform. Technical Report Technical Report No. UCB/EECS-2014-162, University of California at Berkeley, 2014.

[6] ARM. Cortex-M Series. `http://www.arm.com/products/processors/cortex-m/`.

[7] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press Cambridge, 2008.

[8] A. Y. Benbasat and J. A. Paradiso. A compact modular wireless sensor platform. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN)*. IEEE Press, 2005.

[9] J. Beutel, M. Dyer, M. Hinz, L. Meier, and M. Ringwald. Next-generation prototyping of sensor networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*. ACM, 2004.

[10] Q. Cao, D. Wang, T. Abdelzaher, B. Priyantha, J. Liu, and F. Zhao. Energy-optimal batching periods for asynchronous multistage data processing on sensor nodes: foundations and an mPlatform case study. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010.

[11] F. Cassez and K. Larsen. The impressive power of stopwatches. In *CONCUR 2000 - Concurrency Theory*. Springer, 2000.

[12] Crossbow Technology. `http://www.xbow.com`.

[13] H. Dubois-Ferrière, L. Fabre, R. Meier, and P. Metrailler. Tinynode: a comprehensive platform for wireless sensor network applications. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN)*. ACM, 2006.

[14] A. Dunkels, B. Grönvall, and T. Voigt. Contiki – a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks*, 2004.

[15] P. Dutta and D. Culler. Epic: An open mote platform for application-driven design. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 2008.

[16] P. Dutta, S. Dawson-Haggerty, Y. Chen, C.-J. M. Liang, and A. Terzis. Design and evaluation of a versatile and efficient receiver-initiated link layer for low-power wireless. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2010.

[17] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN)*, 2005.

[18] A. El-Hoiydi and J.-D. Decotignie. WiseMAC: An ultra low power MAC protocol for multi-hop wireless sensor networks. In *ALGOSENSORS*, 2004.

[19] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Low-power wireless bus. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2012.

[20] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient network flooding and time synchronization with Glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2011.

[21] Freescale. VF3xxR and MKW2xDx Series. `http://www.freescale.com`.

[22] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2009.

[23] W. P. M. H. Heemels, K. H. Johansson, and P. Tabuada. An introduction to event-triggered and self-triggered control. In *Proceedings of the IEEE 51st Annual Conference on Decision and Control*, 2012.

[24] T. A. Henzinger and J. Sifakis. The discipline of embedded systems design. *Computer*, 2007.

[25] J. L. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 2002.

[26] L. F. Hoesel, S. Dulman, P. J. Havinga, and H. J. Kip. Design of a low-power testbed for wireless sensor networks and verification. Technical report, University of Twente, Centre for Telematics and Information Technology (CTIT), 2003.

[27] A. Jantsch. Models of computation for networks on chip. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD)*, 2006.

[28] A. Jantsch and H. Tenhunen. *Networks on chip*. Springer, 2003.

[29] M. Johnson, M. Healy, P. van de Ven, M. J. Hayes, J. Nelson, T. Newe, and E. Lewis. A comparative review of wireless sensor network mote technologies. In *IEEE Sensors*, 2009.

[30] R. Jurdak, K. Klues, B. Kusy, C. Richter, K. Langendoen, and M. Brunig. Opal: A multiradio platform for high throughput wireless sensor networks. *IEEE Embedded Systems Letters*, 2011.

[31] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: mobile networking for Smart Dust. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*. ACM Press, 1999.

[32] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *Proceedings of the 20th IEEE Real-Time and*

Embedded Technology and Applications Symposium (RTAS), 2014.

[33] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 2003.

[34] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling. Multicore in real-time systems–temporal isolation challenges due to shared resources. In *Proceedings of Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems*, 2013.

[35] Y.-s. Kuo, P. Pannuto, G. Kim, Z. Foo, I. Lee, B. Kempke, P. Dutta, D. Blaauw, and Y. Lee. MBus: A 17.5pJ/bit/chip portable interconnect bus for millimeter-scale sensor systems with 8nW standby power. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2014.

[36] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1997.

[37] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*. Springer, 2005.

[38] C. J. Myers. *Asynchronous Circuit Design*. John Wiley & Sons, 2001.

[39] L. Nachman, J. Huang, J. Shahabdeen, R. Adler, and R. Kling. Imote2: Serious computation at the edge. In *Wireless Communications and Mobile Computing Conference*. IEEE, 2008.

[40] NXP Semiconductors. Cortex-M4 MCUs with Cortex-M0 Co-Processors. `http://www.nxp.com/products/microcontrollers/core/cortex_m0_m4f/`.

[41] B. O'Flynn, S. Bellis, K. Delaney, J. Barton, S. C. O'Mathuna, A. M. Barroso, J. Benson, U. Roedig, and C. Sreenan. The development of a novel minaturized modular platform for wireless sensor networks. In *Proceesdings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN)*. IEEE, 2005.

[42] Oracle Labs. SunSPOT. `http://www.sunspotworld.com`.

[43] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN)*, 2005.

[44] J. Portilla, A. De Castro, E. De La Torre, and T. Riesgo. A modular architecture for nodes in wireless sensor networks. *Universal Computer Science*, 2006.

[45] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2000.

[46] A. Rowe, D. Goel, and R. Rajkumar. FireFly Mosaic: A vision-enabled wireless sensor networking system. In *Proceedings of the 28th IEEE International Real-time Systems Symposium (RTSS)*, 2007.

[47] E. Shih, P. Bahl, and M. J. Sinclair. Wake on wireless: an event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2002.

[48] J. Sifakis. Rigorous system design. *Foundations and Trends® in Electronic Design Automation*, 6(EPFL-ARTICLE-185999), 2012.

[49] J. Stankovic, I. Lee, A. Mok, and R. Rajkumar. Opportunities and obligations for physical computing systems. *Computer*, 2005.

[50] Texas Instruments. F28M3x Series. `http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/c2000_performance/control_automation/f28m3x/products.page`.

[51] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[52] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*. ACM, 2004.