LOCUS: Low-Power Customizable Many-Core Architecture for Wearables

Cheng Tan[1], Aditi Kulkarni[1], Vanchinathan Venkataramani[1],
Manupa Karunaratne[1], Tulika Mitra[1], and Li-Shiuan Peh[2]
[1]School of Computing, National University of Singapore
[2]Department of EECS, Massachusetts Institute of Technology, Cambridge, MA, USA
Email: {tancheng,aditi,vvanchi,manupa,tulika}@comp.nus.edu.sg, peh@csail.mit.edu

ABSTRACT

The requirements' demands of applications, such as real-time response, are pushing the wearable devices to leverage more power-efficient processors inside the SoC (System-on-chip). However, existing wearable devices are not well suited for such challenging applications due to poor performance, while the conventional powerful many-core architectures are not appropriate either due to the stringent power budget in this domain. We propose LOCUS – a low-power, customizable, many-core processor for next-generation wearable devices. LOCUS combines customizable processor cores with a customizable network on a message-passing architecture to deliver very competitive performance/watt – an average 3.1x compared to quad-core ARM processors used in the state-of-the-art wearable devices. A combination of full-system simulation with representative applications from wearable domain and RTL synthesis of the architecture show that 16-core LOCUS achieves an average 1.52x performance/watt improvement over a conventional 16-core shared-memory many-core architecture.

1. INTRODUCTION

Internet of Things (IoT) — a giant, ever-growing network of billions (estimated to be 25 billion by 2020 [4]) of devices embedded within physical objects — is expected to revolutionize our future. Recently, a burgeoning group of these embedded devices, the wearables, is rapidly emerging and bringing new experiences to daily life.

Conventional wearable devices encompass limited functionality (e.g., data collection from on-body sensors, pre-processing the data, temporary data storage), and rely on higher performance endpoints such as mobile phones, gateways or remote servers. However, the increasing demands from customers have been pushing the performance envelope of the wearable devices to provide real-time in-situ computation capability. For example, most smart glasses support augmented reality that requires real-time response [5]. Samsung is offering standalone smart watches with the tag-line "leave your phone at home" [19]. Meanwhile, the development tools of wearables are also rapidly evolving. Many software development kits [2, 6, 20, 21] allow the programmers to create their own applications (e.g., HERE Maps [8], IoT transportation application [49]) on wearable devices.

To meet the growing performance demands, more powerful processors have been deployed inside wearable devices since 2013. Figure 1 shows that the processors used in popular smart watches across different companies are increasing in complexity from single-core ARM Cortex-M to quad-core ARM Cortex-A7, with commensurate performance increase from 150 to 9000 DMIPS. Correspondingly, the typical power consumption rises up to hundreds of milliwatts.

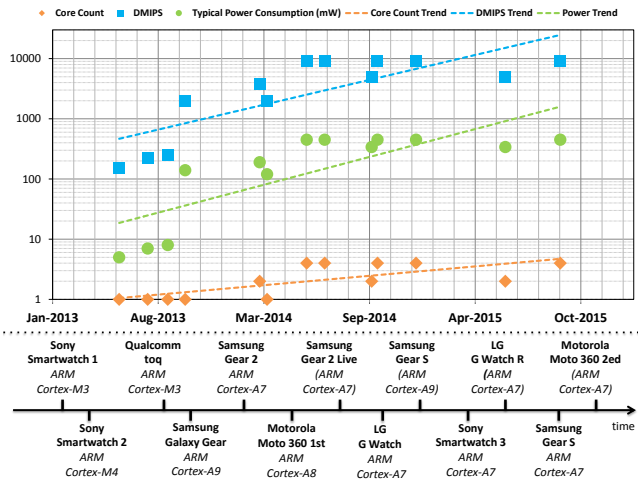

Figure 1: ***Increasing core count in SoCs of smart watches and corresponding power-performance trends***

However, existing wearables still do not provide sufficient performance for emerging IoT applications (detailed in Section 2). Scaling the core count is an obvious option but traditional many-core architectures cannot fit within the stringent power budget of wearables. Application-specific ASIC accelerators improve power-efficiency but are not practical due to the prohibitively high non-recurring engineering (NRE) cost and exacting time-to-market constraints.

There have been recent efforts to design innovative high-performance architectures for low-power sensor nodes [31,34] for health-monitoring applications. But to the best of our knowledge, there have been little attempt to design low-power, high-performance customizable SoCs for wearable devices that work well independent of the application do-

CASES '16, October 01-07, 2016, Pittsburgh, PA, USA
 ISBN 978-1-4503-4482-1/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2968455.2968506

main. We take the first step towards filling this void through LOCUS — a LOw-power, highly CUStomizable many-core architecture that can be universally deployed as a wearable device across diverse application scenarios. We study the characteristics of computational kernels in the wearables domain and to exploit these characteristics to design LOCUS.

LOCUS differs from conventional multi-cores in many key aspects. First, LOCUS uses a lightweight customizable message-passing substrate for data transfers, instead of relying on costly (in area and power) hardware cache coherence prevalent in shared-memory many-cores. Second, LOCUS aggressively customizes the cores and network at run-time in a synergistic and integrated fashion. Frequently occurring instruction sequences in applications are automatically discovered, triggering custom instructions that jointly accelerate computing and communications. These custom instructions configure the processor cores and network at runtime to tailor-fit LOCUS for each specific application, leading to improved performance at lower energy. Together these design decisions and optimizations enable us to design a 16-core LOCUS chip with 6mm×6mm area and average 0.27W power consumption at 400MHz (32nm technology). Even with this low power envelope, the architecture can achieve 1.71x speedup (3.1x gain in performance/watt) over the processor used in the state-of-the-art wearable smart watches. In order to validate the advantage of LOCUS across different architectures regardless of the variations in technology, frequency and core count, we compare with a simulated baseline conventional 16-core shared-memory architecture and observe average 1.52x gain in performance/watt from our evaluation and experiment.

The concrete contributions of this work are the following.

- This is the first work that designs a dedicated many-core architecture for wearable devices embracing parallelism, message passing, and aggressive customizations to realize exceptional performance/watt characteristic.
- We designed LOCUS in RTL, performed RTL synthesis to obtain accurate power, area, timing metrics, and compared LOCUS to quad-core ARM Cortex-A7 processor (used in the state-of-the-art wearable devices today) with real-world wearable applications.
- We implemented LOCUS by modifying the *gem5* [26] architectural simulator and building the message passing library on top of it to enable execution of realistic computation kernels running on wearables for system-level performance evaluation.

The remainder of this paper is organized as follows. Our motivation is presented in Section 2 with an application case study. Section 3 states the related works in literature. Architecture and system designs of our proposed SoC is described in Section 4. The experiment and evaluation of our proposed architecture are illustrated in Section 5 while Section 6 concludes the paper.

2. APPLICATION CASE STUDY

We first present a case study to illustrate the shortcomings of current wearable devices and the potential of the proposed LOCUS architecture. Wearable devices are subject to very stringent power budget. Most wearable devices today comprise of power-efficient processors and run simple computations to minimally process the sensor data, relying instead on more powerful smart phones or the cloud server for handling heavy computing. Transmitting data to the smart phones or cloud server through Bluetooth or wireless networks brings about significant overheads both in terms of increased power consumption (due to wireless communications) and prolonged response time of critical tasks (due to network delay). This has led to increasingly powerful commercial processors being introduced in wearable devices (detailed in Section 3) to enable in-situ processing of the sensor data.

We choose the Dynamic Time Warping (DTW) algorithm as our driving application kernel. DTW is extensively used in speech processing, data mining, gesture recognition, and signal processing [32, 36, 48, 51]. We study the DTW algorithm in relation to a specific IoT application that identifies user contexts like walking, commuting, and waiting for transport [7]. In this application, DTW is used to compare barometer sensor readings with trained barometer signatures to determine the transportation mode. As the barometer detects terrain elevation, and terrain signature of roads remains unchanged, commuters on vehicles that traverse the same road segments (e.g., buses) can be matched to specific routes. The different speed of vehicles versus walking, and the different time signatures of alternative modes such as buses and subway allow DTW to effectively identify the transport mode using the ultra-low-power barometer sensor. We use the raw barometer traces collected by Sankaran et al. [49] through smartphones carried by 13 individuals in 3 countries, gathering 47 hours of transportation traces. Figure 2 shows a DTW based transportation application running on LG Watch Urbane W150 [12]. The application samples barometer sensors at 1Hz frequency to ensure the required sensing fidelity. Hence, the DTW kernel processing per sampling interval has to be limited to 1 second to enable real-time context detection.

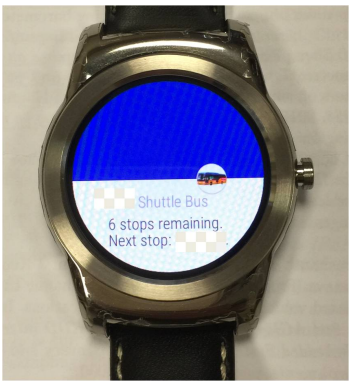

Figure 2: *DTW based transportation application running on LG Watch Urbane W150*

We profile the DTW application on a quad-core ARM Cortex-A7 cluster present in the Odroid XU3 development platform [15], which is similar to the quad-core processor in Qualcomm Snapdragon 400 SoC embedded in state-of-the-art wearable smart watches such as the LG watch above [11–13, 23]. We run Odroid XU3 board at 1.2GHz with 5 volts to emulate the highest frequency of Snapdragon 400. We use Odroid XU3 board (instead of the snapdragon 400 SoC inside LG Watch) as it has in-built power-sensors and allows individual cores to be turned off, enabling us to

obtain the power consumption of the processor rather than the whole SoC and profile the power-performance impact of core counts. Table 1 summarizes the system configuration of this platform.

Number of cores	4 ARM Cortex-A7
L1 I/D Cache	32KB each
L2 Cache	512KB
Frequency	200 - 1400MHz
Voltage	900 - 1050 mV
Process technology	28nm
Operating System	Ubuntu 14.04 LTS

Table 1: *Odroid XU3 system configuration.*

	1-core A7	2-core A7	4-core A7	16-core LOCUS
Meeting Deadline	×	×	×	√
Execution Time (sec)	5.13	3.15	1.56	0.91
Average Power (mW)	164	251	456	266
Energy (mJ)	841	791	711	243
Frequency (MHz)	1200	1200	1200	400
Process technology (nm)	28	28	28	32

Table 2: *Execution time, power, energy consumption of DTW kernel running on different processors*

Table 2 shows the power/energy consumption and execution time of the DTW kernel running on one, two, and four A7 cores. The unused cores are turned off. The average power consumption varies from 164mW $\sim$ 456mW reflecting the ultra-low power constraint faced by wearable processors. In terms of performance, even with four A7 cores active, the computation cannot meet the 1 sec deadline. However, our proposed 16-core LOCUS architecture processes this DTW kernel in 0.913s, meeting the deadline while dissipating 266mW power[1]at 400MHz. Specifically, LOCUS achieves 1.71x speedup with only 58% power consumption (i.e., 2.93x in terms of performance/watt) compared to the quad-core ARM cortex-A7 processor used in state-of-the-art wearable devices. This case study points out the deficiencies of existing wearable devices while confirming the potential of LOCUS in enabling in-situ processing of sensor data at critically low power budget.

3. RELATED WORK

The past three years have witnessed a shift from ultra-low power single-core to more powerful multi-core SoCs for wearable devices. Table 3 shows architectural characteristics of the latest commercial wearable devices from different companies. From this table, we can see that the processor embedded in many state-of-the-art wearable devices [11, 13, 23] is the quad-core ARM cortex-A7 with hundreds of milliwatts typical power consumption.

Sensor nodes utilized in the health care domain have also seen a shift to multi-cores. Authors in [31, 34] propose a multi-core architecture, which performs data processing in the sensor node for quick response time in wearable health monitoring systems. However, popular wearable devices like smart watches are not restricted to only pre-defined applications but can be programmed to support diverse application scenarios. For example, most existing smart glasses support augmented reality with real-time response requirement [5, 17]. Offline navigation applications [8] have also

[1]The power consumption of LOCUS has been projected from 133mW at a frequency of 200MHz (detailed in Section 5.2).

Product (Announced)	SoC	CPU (#core)	Freq (MHz)	Memory	Typical CPU Power (mW)
Google Glass (Apr, 2012)	TI OMAP4430	ARM Cortex-A9 (dual-core)	1000	2GB RAM 16GB Flash	350
Vuzix M100 (Jan, 2013)	TI OMAP4460	ARM Cortex-A9 (dual-core)	1200	1GB RAM 4GB Flash	400
Qualcomm toq (Oct, 2013)	ST STM32	ARM Cortex-M3 (single-core)	200	16MB SRAM 2GB Flash	10
Optinvent ORA-1 (Aug, 2014)	Not available	ARM Cortex (dual-core)	1200	4GB Flash	Not available
Sony Smartwatch 3 (Sep, 2014)	Qualcomm Snapdragon 400	ARM Cortex-A7 (quad-core)	1200	512MB RAM 4GB Flash	450
LG G watch R (Sep, 2014)	Qualcomm Snapdragon 400	ARM Cortex-A7 (quad-core)	1200	512MB RAM 4GB Flash	450
Samsung Gear S2 3G (Aug, 2015)	Not available	ARM Cortex-A7 (dual-core)	1000	512MB RAM 4GB Flash	Not available
Motorola Moto 360 2ed generation (Sep, 2015)	Qualcomm Snapdragon 400	ARM Cortex-A7 (quad-core)	1200	512MB RAM 4GB Flash	450

Table 3: *Specifications of the latest wearable devices*

emerged on standalone smart watches [19]. Many software development kits [2, 6, 20, 21] are available for programmers to create their own applications, such as the transportation application [49] mentioned in Section 2. Health care monitoring apps can also be implemented in wearable devices with appropriate sensors plugged in. The increasing performance demands across diverse application scenarios call for high-performance processors in wearables.

Existing multi-core wearables cannot meet increasing demands of applications especially under real-time scenarios as illustrated in Section 2. At the same time, many-core architectures such as Tilera TILE64 [25], Picochip PicoArray [35], Intel Xeon Phi [10] target domains such as cellular base stations, Internet routers and cloud servers, with high power budget, which is a major obstacle for their deployment in the wearable domain. For example, the Intel Xeon processor consumes 150W [42]. In contrast, LOCUS is carefully designed to operate within the power budget of typical wearable devices while providing far superior performance.

Current wearable devices increasingly leverage heterogeneous architecture integrating lightweight GPUs (e.g., PowerVR, Mali GPU) with general-purpose processor cores together as system-on-chip (e.g., Qualcomm Snapdragon 400 [18], Ineda Dhanush WPU [9]). Although moving the workload of data parallel computing and image processing to the lightweight GPUs likely leads to higher power-efficiency, it is orthogonal to our work. In this paper, we only focus on the CPU architecture.

4. LOCUS SYSTEM ARCHITECTURE

We now describe the system architecture of LOCUS: a low-power, highly-customizable many-core architecture for wearable devices. Figure 3 presents a high-level view of the architecture. The current prototype consists of 16 tiles connected through a customizable mesh network (SMART NoC [27,41]). Each tile contains a customizable CPU (JiTC core [28]), separate instruction and data caches, NIC (network interface controller) plus router, and a lightweight message passing unit (LMPU). The memory controller is connected to the router inside the first tile. LOCUS employs aggressive customization at multiple levels to deliver very competitive performance/watt: (a) *Communication*: Cus-

tom message-passing instructions in place of generic shared-memory to minimize communication cost, (b) *Interconnect:* Custom single-cycle datapath between any two communicating cores through SMART NoC, (c) *Compute cores:* Custom instructions extend the ISA (instruction-set architecture) at runtime with frequently occurring computational patterns through JiTC core, and (d) *Integrated compute-communicate customization:* Specialized custom instruction pairs at two communicating cores seamlessly combine compute and communication customization. We will next detail each component of LOCUS before putting it all together, showing how LOCUS enables integrated customization of compute and communications for specific applications, delivering the target power-performance of wearables.

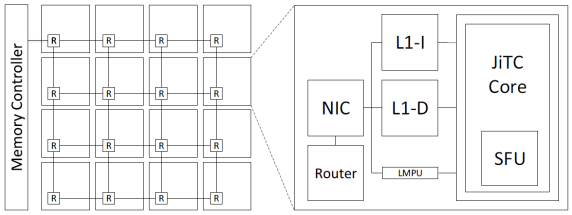

Figure 3: ***Overview of LOCUS architecture***

4.1 Lightweight Message-Passing (LMP)

Conventional shared-memory many-core architectures suffer from high energy and latency overheads in maintaining coherence among the private caches [56]. These overheads motivated us to replace the shared-memory model with a lightweight message-passing (LMP) mechanism in LOCUS. Explicit message passing leads to fast communications and alleviates unnecessary data transfers. Moreover, elimination of hardware cache coherence management and directory structures reduces overall power consumption, saves precious on-chip resources, and provides better scalability.

A number of commercial many-core architectures support message-passing including IBM Cell [37], MIT Raw [52], Intel SCC [39], Epiphany [38], and MEDEA [53]. Some of these architectures provide sophisticated software support such as large message-passing library code and complex interrupt handlers [46, 47] for compatibility with MPI codebase [14] or high performance computing. These factors deteriorate performance and increase power consumption. Instead, in the context of wearables, we opt for hardware-assisted lightweight message-passing to alleviate the software overhead and minimize communication cost.

As mentioned earlier, each tile in LOCUS features an LMPU that connects the CPU to the NIC. The LMPU comprises a message buffer [40] to temporarily store data received from remote cores. The message buffer is designed as a 16-entry fully-associative queue where each entry stores 32-bit data along with the source ID. The message buffer can be searched for data received from a source core (if any) within a cycle. The LMPU also maintains a 2-bit status for each destination core to record the current communication status, which indicates whether a send or receive request is issued and whether the data sent got buffered on the receiver side or not. In LOCUS, we support both *register-to-register* and *cache-to-cache* communications for different granularity of data.

4.1.1 Register-to-Register Communication

We introduce two new instructions **RLD (remote load)** and **RST (remote store)** in the compute core to support register-to-register communication between the cores. Programmers write applications using LMP Application Programming Interface (API). For example, in Figure 4, Core 0 sends data to Core 1 by calling LMP API `Send` function with a pointer to the data variable and destination core ID as arguments. Similarly, Core 1 receives data from Core 0 by calling `Recv` function with a pointer to the variable to receive data and source core ID as arguments. The `Send` and `Recv` functions are converted to assembly instructions `RST` and `RLD` by our modified compiler.

The `RST` instruction has two source registers: data and destination core ID. The register data is sent to the destination core and is accepted by the corresponding `RLD` instruction whose source register represents the source core ID and a destination register accepts the data from the source core. The CPU treats `RLD/RST` instructions as normal memory instructions and sends them to the Load Store Queue (LSQ). However, the LSQ activates the LMPU rather than L1 cache controller for `RLD/RST` instructions. LMPU is also activated when the NIC receives data that has been pro-actively sent by a remote core before being requested by current core.

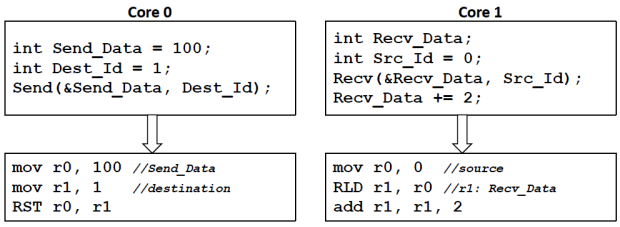

Figure 4: ***LMP API and remote load/store***

Remote Store: A `RST` instruction triggers the flow of the data (4-bytes) from LSQ to LMPU to NIC to generate a network flit that is sent to the destination. If the destination core has already executed the corresponding (`RLD`) instruction, then that instruction would have triggered a receive request earlier to the source duly recorded in the status register of the source LMPU. In this case, the source can proceed to execute subsequent instructions. Otherwise, the source waits for acknowledgment. The LMPU at the destination core sends an acknowledgment only if the data can be buffered. If not, it discards the incoming request. This makes the source wait till the destination executes an explicit receive (`RLD`) forcing the source to resend the flit but this time without waiting for an acknowledgement.

Remote Load When a remote load instruction is executed on the destination core, it first checks if the data is already available in the message buffer, i.e., the source has already sent the data. Otherwise, the LMPU sends a remote load request to the source and waits for the data to arrive. The source immediately sends the data if the corresponding `RST` instruction had already executed. Otherwise, the status bits at source LMPU are updated. When the corresponding `RST` instruction is executed by the source, the LMPU checks the status bits and immediately sends the data as mentioned before. Programmers should thus post remote loads before remote stores as much as possible to prevent the buffer from getting full.

Eliminating Deadlocks In our communication protocol,

a send will not complete until the data is either buffered or read in the destination. Similarly, a receive will not complete until the data becomes available in the message buffer. The buffering of messages, if necessary, avoids deadlocks, while the proactive remote loads avoids buffering latency for faster transfers.

4.1.2 Cache-to-Cache Communication

Besides the asynchronous word-length data transfer, LOCUS supports a *synchronous* cache-to-cache communication for *bulk transfer* through a pair of new instructions **VRLD (varisized remote load)** and **VRST (varisized remote store)**. For example, in Figure 5, Core 1 receives data from Core 0 by calling LMP API `VRecv` function with starting address, data length, and source core ID as arguments. Correspondingly, Core 0 sends data to Core 1 by calling `VSend` function with starting address, data length, and destination core ID as arguments. The `VSend, VRecv` functions are converted to assembly instructions `VSET` followed by `VRST` or `VRLD` by our modified compiler. Specifically, the `VSET` instruction indicates the (`data_size` in bytes) to the cache controller. Note that `VRST` instruction with data length less than a single cache line size still sends the whole cache line and hence the compiler should ensure proper data alignment to avoid spurious data transfer. Then, `VRLD` triggers the cache controller on Core 1 to issue a varisized remote load request carrying the receiver data address (`Recv_Array`) to Core 0 (`Src_Id`). Once the `VRLD` request arrives and `VRST` is executed on Core 0, its cache controller forwards the cache lines indicated by *Send_Array* address to Core 1 (`Dest_Id`). The cache lines are bundled up as a network package with `Recv_Address` and sent into the NoC continuously[2] until the preset data size is reached or exceeded. Similarly, Core 1 finishes receiving once the size of the received data equals to or exceeds the preset data size `data_size`.

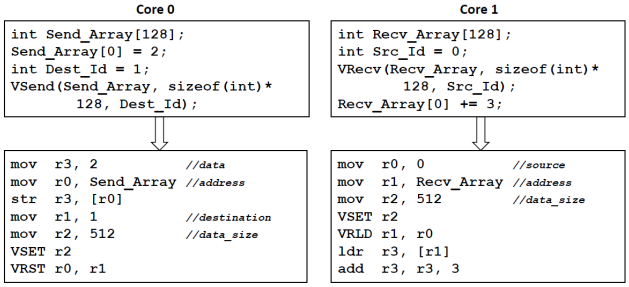

Figure 5: ***LMP API and custom cache-to-cache data-transfer instructions***

During cache-to-cache data transfer, if there is a cache line miss in source core, a main memory access request will be sent to the memory controller with the *requester* being set to the destination core and the destination address set to `Recv_Array`. Therefore, the memory controller will directly forward the data to destination core after fetching from main memory. Once the data arrives at destination, it will update the cache line according to `Recv_Array` address. Note that the replacement policy would be triggered first if there is no space available in the cache.

[2]All the data to be transferred should be within a single page (the alignment is done by gcc) in a system with virtual address support to avoid re-mapping from virtual to physical address.

4.2 Customizable Interconnect

In LOCUS, we leverage SMART NoC [27, 41] to achieve single-cycle data-path between source and destination after a custom message passing communication is established between any pair of source and destination cores on-chip. SMART achieves this by replacing clocked link drivers at each router along the path with clockless repeaters, enabling a flit to traverse multiple hops in one cycle (without buffering at intermediate routers). A custom single-cycle path is created automatically where available, providing the illusion of a dedicated interconnect atop a shared NoC.

Once a custom message-passing instruction (`RLD/RST` or `VRLD/VRST`) is executed, the corresponding communication will launch into the SMART NoC. A SMART-hop starts from the source NIC, where flits are buffered. A SMART-hop setup request (SSR) is sent by the source NIC via dedicated repeated wires (which are inherently multi-drop) to all routers within 4 hops. All intermediate routers arbitrate between the SSRs they receive to determine if flits can zoom through without stopping, or if flits should be buffered due to contention. In this way, remote read/writes can be almost as fast as local loads/stores in LOCUS, significantly improving the overall power-efficiency especially for communication-intensive workloads.

4.3 Customizable Compute Cores

The wearable device can be deployed in many different application domains. Therefore, in-situ data processing on wearables exhibit vastly different computational patterns. Adapting the architecture to exploit diversity within and across applications can significantly improve performance-power trade-offs. But due to high non-recurring engineering cost of SoC devices, it is not feasible to design customized circuits for specific application scenarios. We leverage Just-in-Time Customizable (JiTC) cores [28] to reconcile the conflicting demands of performance and flexibility. Each JiTC core contains a *Special Functional Unit (SFU)* in conjunction with a simple processor pipeline. The SFU can be configured at runtime to execute custom instructions that accelerate commonly occurring computational patterns in an application. As the SFU for each JiTC core can be specialized with different custom instructions, LOCUS can be transformed into a heterogeneous many-core architecture where distinct computational workloads are mapped onto different tiles and accelerated by different SFUs.

Each JiTC core is very power-efficient, featuring a simple 5-stage, single-issue in-order pipeline implementing 32-bit ARM ISA and integrated with an SFU. The JiTC compiler [28] can automatically identify frequently occurring computational patterns within an application. Each of these patterns can have at most 4 inputs and 2 output operands. The selected computational patterns are converted into custom instructions and added to the baseline ISA. The custom instructions are issued to the SFU instead of the execution unit inside the in-order pipeline. The SFU consists of a complex functional unit in parallel with two basic functional units without extending the critical path. Each basic functional unit consists of an ALU followed by a shifter while the complex functional unit has an additional multiplier. The SFU is able to execute most computational patterns with various compositions of operations in one clock cycle by setting the control bits for each internal functional unit and the MUXes that connect the different internal components.

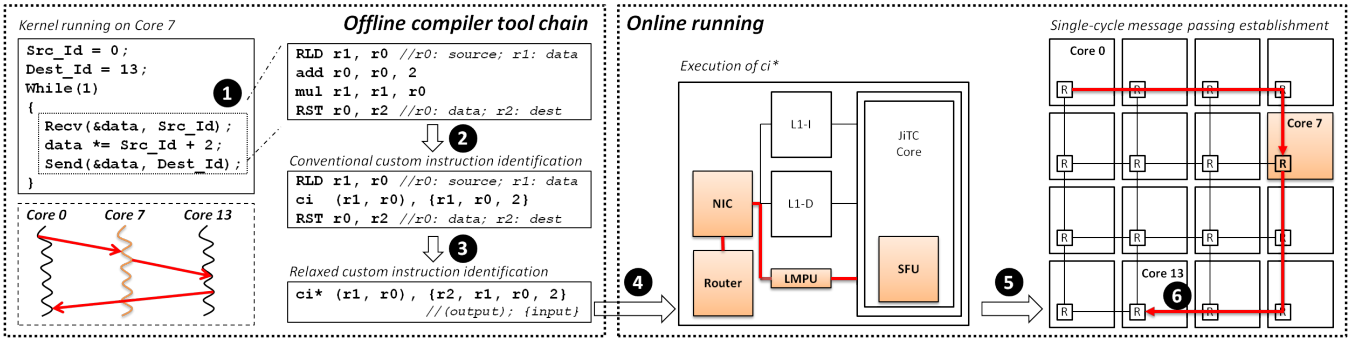

Figure 6: ***Walkthrough example illustrating LOCUS' integrated, customizable compute and communications***

The remaining complex custom instructions may execute in two or more cycles. The control bits corresponding to each custom instruction are stored in a control memory and is indexed by a subset of the opcode of the custom instructions. Before each application initiates execution, the control memory needs to be loaded. As the subset of custom instructions selected varies for different applications, the content of control memory is unique for each application. In other words, the JiTC architecture achieves customization by changing the content of the control memory and thereby instantiating different custom instructions per application. The size of the control memory is set to be as big as 32KB in [28]. However, our evaluation shows that the maximum number of identified static custom instructions among all kernels is no more than 30, which is far fewer than 1024 that is theoretically supported by [28]. Furthermore, only the identified computational patterns that can be executed within single cycle, which occupies 90% of all identified patterns, are finally selected as custom instructions. Therefore, we chose a small 256B control memory for LOCUS.

4.4 Compiler Tool Chain

In order to support LOCUS architecture, we implement an automated compiler tool chain integrated with a modified GNU Assembler. Given a multi-threaded application written in C or C++ with the LMP API, the tool chain first converts LMP functions to appropriate message-passing instructions. This is done using inline assembly in LMP API implementation. The compiler then detects the 'hot' (frequently occurring) basic blocks through profiling. The data-flow graphs of these 'hot' basic blocks are analyzed to identify all the potential candidate custom instruction patterns [55]. A subset of these candidate patterns is selected for implementation as custom instructions and mapped onto the SFU through a greedy heuristic mapping algorithm [28,44], while the control bits to be loaded into control memory for the mapping to SFU are generated in parallel. The tool then replaces each occurrence of a selected pattern in the code (consisting of a sequence of instructions from the base ISA) with the corresponding custom instruction in assembly, and generates the binary executable through the modified GNU Assembler.

4.5 Integrated Compute-Communication Customization

Customizations in different layers are seamlessly combined and adaptively optimized to maximize the power-efficiency in LOCUS. Conventionally, memory and data transfer (load/store) operations are not included within custom instructions due to unpredictable data access latency and implicit memory dependencies. Fortunately, the custom message-passing instructions (`RLD/RST`) of LOCUS with fixed-access latency to the LMPU permits the inclusion of communications within custom instructions. We use a simple example to illustrate this optimization. As shown in Figure 6, ❶ the frequently executed basic block running on Core 7 is first detected as 'hot' basic block by the compiler tool chain according to the profile information during offline analysis. Then, ❷ the frequently occurring computation pattern (*add* and *mul*) within the 'hot' basic block is identified as a custom instruction candidate (*ci*). ❸ By relaxing the constraint of excluding load/store, the message-passing instructions are also included in generating *ci** that contains all four original instructions. As a result, a single-cycle input-compute-output processing flow is encapsulated within a single custom instruction. ❹ Once the custom instruction (*ci**) is selected, the compiler generates the executable file and the corresponding control bits for JiTC core. At runtime, instead of the execution unit inside the pipeline, the SFU takes responsibility to execute the computation operations (*add* and *mul*) inside the custom instruction (*ci**). The remote read and store operations will trigger the NIC to launch the communication request. ❺ Once the custom message passing communication is launched, the SMART router establishes single-cycle data-path from the source to the destination cores by bypassing the buffers inside the intermediate routers. ❻ When the data finally arrives at its destination, the program execution on Core 13 will resume forward progress.

5. EXPERIMENTAL EVALUATION

This section presents a detailed experimental evaluation of LOCUS architecture for its suitability in wearable devices. We employ a combination of RTL synthesis and high-level architectural simulations with representative kernels running on wearable devices for this evaluation.

5.1 Simulation Environment

We use the *gem5* multi-core architectural simulator [26] for performance evaluation of LOCUS. Our baseline is a directory-based MESI cache-coherent shared memory architecture consisting of 16 in-order ARM cores connected in a conventional 2D mesh NoC (Garnet [24]). The baseline is only implemented and simulated in gem5, and not in RTL. This is to mitigate the significantly higher implementation and simulation time complexity of RTL simulations, as gem5 architectural simulations permit much wider design space exploration.

The *gem5* simulator is modified to model LOCUS message-passing architecture with 16 JiTC cores connected by SMART NoC. ARM cores are chosen due to their power efficiency. We extend the ISA to support message passing instructions (`RLD/ RST, VRLD/VRST, VSET`) and other custom instructions executed by SFUs in JiTC cores. We integrate the JiTC compiler tool chain with the modified GNU Assembler for ARM ISA to support LOCUS by identifying custom instructions based on profiling information, replacing the native instructions of the selected computational-communication patterns with custom instructions in the assembly, and generating the executable and the control memory configuration data corresponding to the selected instructions. Light-weight message-passing (LMP) API is implemented based on MPICH library [14]. We can support cycle-accurate execution of applications parallelized using LMP API and extended with custom instructions.

Detailed parameters of the simulated system are listed in Table 4. Both LOCUS and the baseline have similar configuration except that LOCUS does not have directory and its network is replaced by SMART.

Cores	16 ARM in-order (single-issue) cores 400MHz
L1 Cache	split I & D, 8KB, 2-way, 64B block, LRU, 1-cycle access latency
Directory	MESI coherence, single slice, 6-cycle access latency
Network	2-D Mesh, 16B-flit, 1/5-flit control/data packets, 5-stage router, 1-cycle link
Memory	512MB, 100-cycle DRAM access latency

Table 4: ***RTL and simulation parameters for LOCUS and baseline share-memory multi-core.***

5.2 Workloads

As there does not exist any benchmark suite for wearables, we choose a set of representative kernels that are widely used in wearable devices.[3] Dynamic time warping (**DTW**) is used in speech processing, data mining, gesture recognition and signal processing. We use a specific variant of parallel DTW enabling higher parallelism [45]. Navigation applications, especially offline navigation algorithms that do not require tethering to the phone or the Internet, is becoming increasingly critical on wearables [8]. **A Star** search algorithm is typically used as a navigation kernel [16] and we implement a parallel version [33]. Personal health monitoring systems can offer a cost-effective healthcare solution and the electrocardiogram (**ECG**) delineation is a typical and essential application in this domain. We implement an **ECG** R-peaks detection [3] for our evaluation. Encryption/Decryption is another frequently used kernel in wearables for secure data communications. We use **AES Encrypt** and **AES Decrypt** as the representative kernels. Image processing is increasingly applied in wearable devices, especially augmented reality glasses. Kernels like **2D Convolution** and **Histogram** are used in such scenarios for sharpening, smoothening and enhancing images. As wearable devices have limited storage, sensor data needs to be compressed before storage. We use the **Haar Transform** variant of Discrete Wavelet Transform for compressing sensor data. We also implement a Support Vector Machine (**SVM**) kernel as wearable devices are extensively used for classifying patterns based on sensor data. All the workloads are manually parallelized. Table 5 summarizes the representative kernels used in our evaluation along with the input size, communication patterns, and the parallelism. We define pipeline communication pattern as one where a core receives data from a previous core before processing (i.e., producer-consumer relationship). In scatter-gather communication pattern, one core works as the master to launch processing tasks onto the other cores, then collects the data from them at the end. The 2D Convolution and Haar transform kernels do not have communications between threads.

[3]We make these kernels publicly available: https://github.com/iot-locus/kernels

Category	**Kernel**	**Problem scale**	**Typical communication pattern**	**Parallelism (speedup)**
Pattern maching	DTW	5000 x 5000	pipelined	13.2
Navigation	A Star	3770 nodes	scatter-gather	12.8
Health Monitoring	ECG	18700	scatter-gather	10.9
Encryption	AES Encrypt	4000 bytes	pipelined	11.6
	AES Decrypt	4000 bytes	pipelined	11.1
Image Processing	Histogram	1024 x 1024	scatter-gather	13.8
	2D Convolution	1024 x 1024	None	15.7
Machine learning	SVM	12224 1 x 4 Support Vector	scatter-gather	13.8
Compression	Haar Transform	1024	None	15.4

Table 5: ***Representative wearable application kernels.***

We also implemented two applications: combo1 (SVM + AES) and combo2 (AES Decrypt + DTW + AES), which make use of the above mentioned kernels. Combo1 is used to classify sensor data like images as anomalous or not. It runs SVM Machine Learning kernel to recognize the anomalous image, then encrypts it for future references. Combo2 is used in context-detection; it gets encrypted barometer sensor data as input, decrypts it, runs the DTW algorithm to identify the context, then encrypts the output before sending it to smartphone or cloud storage.

We first implement each parallel kernel using POSIX threads for our baseline shared-memory architecture. Table 5 shows the speedup for the 16-core shared-memory architecture baseline and confirms the highly parallel nature of these kernels. Note that the serial performance is simulated in *gem5* with the same processor configuration but with a single core. The speedup of AES kernels is slightly lower due to the frequent lock contention. The speedup of 2D convolution and Haar transform are very close to linear, as they are implemented in a totally data parallel fashion, where 16 cores could simultaneously execute without communication.

For evaluation of LOCUS, we implement all the kernels using message passing programming model with the LMP API. The communications in A Star and SVM are implemented through cache-to-cache data transfer as transfer size in each communication is always larger than 4-byte register-to-register data transfer. The integrated compute-communication custom instructions are leveraged in many of the kernels. Note that we impose the restriction of at most 4 input operands and 2 output operands per candidate pattern [29, 30, 54] during custom instruction identification.

5.3 Synthesis Results

We implement LOCUS architecture in RTL. We leverage open-source ARM Amber core [1], and integrate it with an SFU to form the LOCUS JiTC processing core inside each tile, with the SFU running in parallel with the execute stage of Amber. Each tile contains 2-way associative 8KB instruction and data caches with 64 bytes of cache lines for which we use 64x24 and 64x512 SRAM blocks per set for tag and data, respectively. The tile also contains a SMART router with a 5x5 crossbar. A NIC is used to reassemble flits and disassemble packets and an LMPU containing 16-entry buffer stores data received from the other cores. In our design, Tile0 is also connected to a memory controller.

The design is implemented in Verilog and simulated with Synopsys VCS-MX. Synopsys 32nm generic standard cell and SRAM libraries are used to synthesize, place and route our design in Design Compiler and IC Compiler to estimate area, timing and power numbers.

Timing analysis. Our design synthesizes at 200MHz with an operating voltage of 0.95V and 25°C temperature. The critical path is in the Amber core with a cumulative path delay of 4.88ns which goes through the write_back output (0.11ns) → execute (1.9ns) → barrel_shifter (0.59ns) → ALU (2.17ns) → execute output (0.11ns). Note that on a commercial 28nm FD-SOI process, our design can synthesize to 1GHz, demonstrating that higher clock frequency can be attained, though it is not necessary for our applications. We chose the Synopsys generic 32nm PDK, and synthesizing LOCUS on this PDK achieved a maximum frequency of 200MHz.

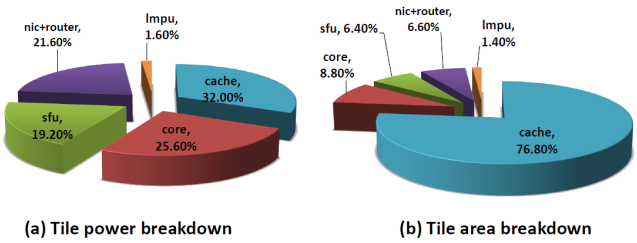

Figure 7: ***Power and area breakdown of LOCUS tile.***

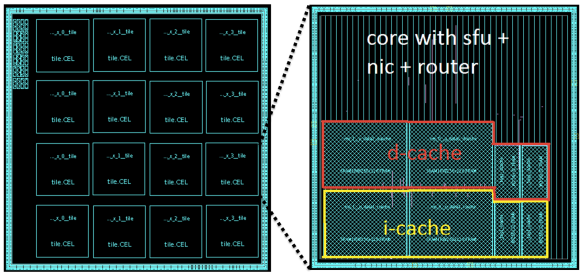

Figure 8: ***Layout of LOCUS many-core architecture.***

Power analysis. The power consumption of LOCUS is derived from RTL simulation fed with instruction and memory traces generated from running benchmark applications on *gem5*. The switching activity information file generated in Synopsys VCS-MX is fed as input to Synopsys DC and ICC to derive accurate power estimates. The power consumption for the entire chip is 133 mW at 200MHz (estimated 266mW at the target frequency of 400 MHz). Figure 7 shows the power breakdown of the LOCUS chip, where the UnCore (NIC, router) takes up <25% of total power, a significantly lower fraction than most shared memory multicores whose Uncore comprise shared last-level caches, coherence controllers and interconnect of the UnCore.

Area analysis. LOCUS chip area is 6 mm x 6 mm, carried through place-and-route to layout, which satisfies the area requirement of the wearables' SoC like the Qualcomm Snapdragon 400 ($40{\sim}50mm^2$ [22]) including a quad-core ARM cortex-A7 processor. The layout of LOCUS is illustrated in Figure 8. The area breakdown in Figure 7 shows Uncore taking up just ∼8% area. Note that the maximum distance in LOCUS for routing is 12 mm which is less than the maximum SMART routing distance constraint [41]. Hence, the single cycle routing in LOCUS is guaranteed.

5.4 Comparison with Processors in State-of-the-art Wearable Devices

We illustrate the potential of LOCUS by first comparing it with quad-core ARM Cortex-A7 processor utilized in state-of-the-art wearable devices. Figure 9 shows the speedup and normalized power consumption across kernels running on LOCUS with respect to the quad-core ARM Cortex-A7 running at 1.2GHz. Note that the execution time for LOCUS is collected from *gem5* running at 400MHz while its power consumption is obtained from the RTL simulation. We use Odroid XU3 board to obtain the execution time and power consumption of state-of-the-art quad-core Cortex-A7 processor (see Section 2). LOCUS achieves an average 1.71x speedup while dissipating only 55.2% power across all kernels (3.1x in terms of performance/watt) compared to 4-core Cortex-A7. The speedup of 2D convolution and Haar transform are better than the others thanks to their higher parallelism.

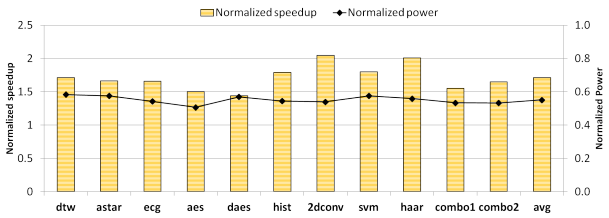

Figure 9: ***Normalized speedup and power consumption of LOCUS across kernels compared to 4-core ARM Cortex A7 in state-of-the-art wearable devices.***

5.5 Comparison with 16-Core Shared Memory Architecture

In order to verify the advantage of LOCUS across different architectures regardless of the variations in technology, frequency and core count, a baseline of conventional shared-memory architecture with the same scalability (16-core) is compared. LOCUS achieves impressive improvement in terms of performance/watt. The best case improvement reaches 2.54x and the average is 1.52x (calculated according to Figure 10 and Figure 11). We will discuss the impacts of different components of LOCUS on performance and power, respectively.

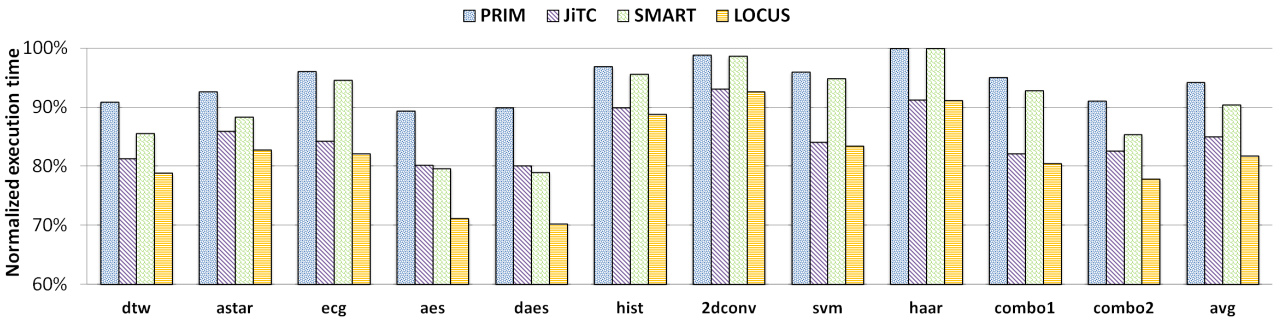

Figure 10: ***Normalized execution time with respect to the 16-core shared memory baseline***

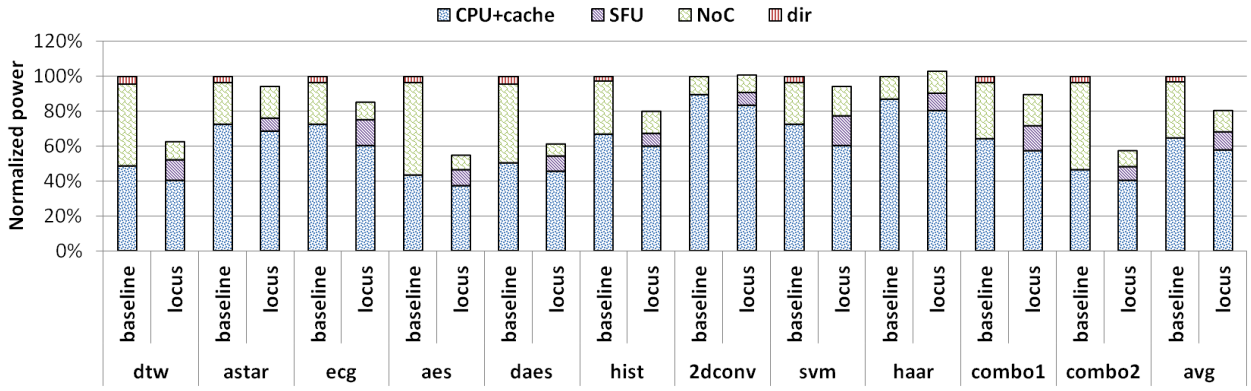

Figure 11: ***Normalized power breakdown with respect to the 16-core shared memory baseline***

First, we evaluate the execution time reduction (Figure 10) for LOCUS with four different configurations (**PRIM**: LOCUS with both JiTC core and SMART NoC disabled; **JiTC**: LOCUS with only JiTC core enabled; **SMART**: LOCUS with only SMART NoC enabled; **LOCUS**: LOCUS with both JiTC core and Smart NoC enabled) compared to our baseline directory-based shared memory many-core architecture. LOCUS achieves an average 18% reduction in execution time with SMART NoC and JiTC core compared to the baseline.

The contributions of the JiTC core and the SMART NoC are observed to be different in each of these kernels. A 10% reduction in execution time could be observed when using JiTC core with computationally expensive kernels like ECG, AES, SVM, and Combo1 (includes SVM). The SMART NoC further reduces the execution time by more than 5% for communication intensive kernels like DTW, A Star, AES and Combo2. The variation in performance for the rest of the kernels is attributed to less or no communication. The combination kernels achieve significant performance gain. Still, this gain is less than the sum of their individual gains, which is due to communication bottlenecks present in the interaction among kernels.

Next, we evaluate the power behavior of LOCUS compared to 16-core shared-memory baseline. The power consumption of CPU and NoC are obtained by feeding the configurations and statistics from *gem5* to McPAT [43] and DSENT [50], respectively. Figure 11 shows that for most kernels, LOCUS consumes less power than the 16-core shared-memory baseline, because it eliminates the coherence traffic and bypasses the routers in NoC. Even for the non-communicating kernels (i.e., 2D Convolution and Haar Transformation) whose power saving in CPU and NoC cannot compensate the power losing in SFU, the increased power consumption is negligible.

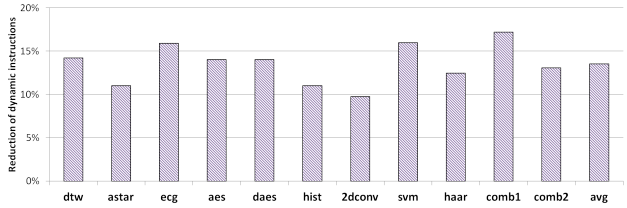

Figure 12: ***Reduction of dynamic instructions in LOCUS after replacing frequently occurring compute-communication patterns with custom instructions***

Figure 12 shows the reduction in number of dynamic instructions in LOCUS after replacing the frequently occurring computational patterns with custom instructions. The less instructions are executed, the less is the execution time (Figure 10). On an average, the number of dynamic instructions decreases by 13.6% across different representative kernels.

Finally, we evaluate the number of flits transferred in the NoC and their corresponding latency. As seen in Figure 13, LOCUS saves more than 50% of flits transferred in the NoC when compared to the baseline. AES and AES Decrypt save most flit transfers due to explicit message passing, which effectively alleviates the lock contention happening in the shared memory baseline. LOCUS cannot save flits transferred in NoC for 2D Convolution and Haar Transformation because there is no communication inside these data parallel kernels. The flits transferred for A Star and SVM in LO-

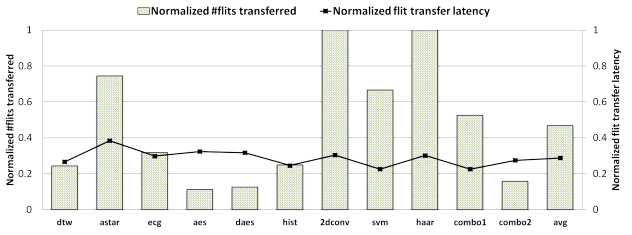

Figure 13: ***Normalized number of flits and transfer latency with respect to the 16-core shared memory baseline***

CUS are not significantly decreased as their communications are implemented in the synchronous cache-to-cache transfer fashion. Moreover, the flit transfer latency on LOCUS decreases by 70% on an average compared to the baseline due to the single-cycle path generated by SMART NoC.

6. CONCLUSION

In this paper, we propose LOCUS — a low-power, customizable, many-core processor for next-generation wearable devices. Instead of relying on the smart phones, gateways or cloud servers, LOCUS can satisfy performance requirements of applications in-situ under the typical power budget of hundreds of milliwatts, to improve the real-time processing capability and sensing fidelity. By using lightweight message-passing, a customizable interconnect and customizable compute cores, LOCUS achieves an average 3.1x performance/watt improvement compared to the quad-core ARM processor used in the state-of-the-art wearable devices. A combination of full-system simulation and RTL synthesis of the architecture with representative wearable applications shows that LOCUS achieves an average 1.52x performance/watt gain over a conventional shared-memory many-core architecture with the same core count.

ACKNOWLEDGMENTS. This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2014-T2-2-129.

7. REFERENCES

[1] Amber Arm-Compatible Core. http://goo.gl/jshd3q.
[2] AR Glasses SDK. http://goo.gl/o9Y5YM.
[3] ECG Processing – R-Peaks Detection. http://goo.gl/oybn8c.
[4] Gartner Inc. http://goo.gl/tvinzf.
[5] Google Glass. https://goo.gl/2VDMyO.
[6] Google Glass SDK. https://goo.gl/jWeUh5.
[7] Google's Fused Location API. https://goo.gl/fackd8.
[8] HERE Maps. http://goo.gl/lVPqux.
[9] Ineda Dhanush WPU. http://goo.gl/SFml7h.
[10] Intel Xeon Phi. http://goo.gl/8jxtzr.
[11] LG G Watch. http://goo.gl/5BZ5zD.
[12] Lg Watch Urbane w150. http://goo.gl/qg76vg.
[13] Moto 360. http://goo.gl/N1jquY.
[14] MPICH. https://www.mpich.org/.
[15] Odroid-XU3. http://goo.gl/vhPocF.
[16] Offline Navigation. http://goo.gl/Bmeljs.
[17] ORA by Optinvent. http://optinvent.com/.
[18] Qualcomm Snapdragon 400. https://goo.gl/aja771.
[19] Samsung Gear S. http://goo.gl/aE6ApL.
[20] Samsung Gear SDK. http://goo.gl/cT4qXJ.
[21] SmartWatch 2 APIs. https://goo.gl/IBGTmg.
[22] Snapdragon 400 Chip Cost. http://goo.gl/YAIqzJ.
[23] Sony SmartWatch 3. http://goo.gl/qrV8ux.
[24] N. Agarwal et al. GARNET: A detailed on-chip network model inside a full-system simulator. In *ISPASS'09*.
[25] S. Bell et al. Tile64-processor: A 64-core soc with mesh interconnect. In *ISSCC'08*.
[26] N. Binkert et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.
[27] C.-H. O. Chen et al. SMART: a single-cycle reconfigurable NoC for SoC applications. In *DATE'13*.
[28] L. Chen et al. A just-in-time customizable processor. In *ICCAD'13*.
[29] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO'04*.
[30] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *ISCA'05*.
[31] F. Conti et al. PULP: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision. *Journal of Signal Processing Systems*, 2015.
[32] A. Corradini. Dynamic time warping for off-line recognition of a small gesture vocabulary. In *Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems, 2001.*
[33] Z. Cvetanovic and C. Nofsinger. Parallel astar search on message-passing architectures. In *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on*, volume 1, pages 82–90. IEEE, 1990.
[34] A. Y. Dogan et al. Multi-core architecture design for ultra-low-power wearable health monitoring systems. In *DATE'12*.
[35] A. Duller et al. Parallel processing-the picoChip way. *Communicating Processing Architectures*, 2003.
[36] A. Efrat et al. Curve matching, time warping, and light fields: New algorithms for computing similarity between curves. *J. Math. Imaging Vis.*
[37] M. Gschwind et al. Synergistic processing in Cell's multicore architecture. *MICRO'06*.
[38] L. Gwennap. Adapteva: More flops, less watts. *Microprocessor Report*, 6(13):11–02, 2011.
[39] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC'10*.
[40] L. Huang et al. Accelerating NoC-based MPI primitives via communication architecture customization. In *ASAP'12*.
[41] T. Krishna et al. Breaking the on-chip latency barrier using SMART. In *HPCA'13*.
[42] B. Li et al. The power-performance tradeoffs of the Intel Xeon Phi on HPC applications. In *IPDPSW'14*.
[43] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO'09*.
[44] L. McMurchie and C. Ebeling. PathFinder: a negotiation-based performance-driven router for FPGAs. In *FPGA'95*.
[45] M. Müller. Dynamic time warping. *Information retrieval for music and motion*, 2007.
[46] M. Ohara et al. MPI microtask for programming the Cell broadband engine processor. *IBM Systems Journal*, 2006.
[47] J. Psota and A. Agarwal. rmpi: Message passing on multicore processors with on-chip interconnect. In *HiPEAC'08*.
[48] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 1978.
[49] K. Sankaran et al. Using mobile phone barometer for low-power transportation context detection. SenSys'14.
[50] C. Sun et al. DSENT-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *NoCS'12*.
[51] C. Tappert et al. The state of the art in online handwriting recognition. *Pattern Analysis and Machine Intelligence*, 1990.
[52] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *MICRO'02*.
[53] S. V. Tota et al. MEDEA: a hybrid shared-memory/message-passing multiprocessor noc-based architecture. In *DATE'10*.
[54] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *DAC'04*.
[55] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES'04*.
[56] J. Zebchuk et al. A tagless coherence directory. In *MICRO'09*.