Journal of Signal Processing Systems
https://doi.org/10.1007/s11265-021-01682-y

Exploring Domain-Specific Architectures for Energy-Efficient Wearable Computing

Dhruv Gajaria[1] · **Tosiron Adegbija**[1]

Received: 1 February 2021 / Revised: 23 April 2021 / Accepted: 2 July 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract

This paper explores the use of domain-specific architectures for energy-efficient and flexible computing of a variety of workloads, including signal processing applications, in wearable devices. As wearable devices become more popular, and with growing consumer demands, these devices are expected to run a wide range of increasingly complex workloads. A general-purpose solution for wearable computing (e.g., microcontrollers and microprocessors) affords high flexibility, wherein a wide range of applications can be run, but offers mediocre performance and may result in high energy and area overheads. On the other end of the computing flexibility spectrum, application-specific integrated circuits (or accelerators) may optimize a specific algorithm, resulting in inflexible computing and under-utilization of computing resources. Domain-specific architectures (DSAs) provide a happy medium of computing flexibility. DSAs focus on *doing a few things*—i.e., satisfying the computing requirements of a set of domain workloads with execution similarities—*extremely well*. As such, DSAs maximize resource usage and achieve substantial performance and energy benefits for a variety of applications. In this work, we first analyze wearable workloads to identify their execution patterns, data movement characteristics, execution bottlenecks, and similarities. Thereafter, we explore various DSA design schemes to meet the increasing processing requirements of wearable workloads, within the typically stringent design constraints of wearable devices. We analyze the performance, energy, and area tradeoffs of the different DSA design schemes in comparison to multiple state-of-the-art architectures, and show, through experimental results, that DSAs offer much promise for flexible, low-overhead, and energy-efficient wearable computing.

Keywords Domain-specific architectures · Wearable devices · Energy efficient · Internet of Things (IoT) · Wearable computing

1 Introduction and motivation

In today's digital and connected world, consumer devices such as wearables have become some of the fastest growing Internet of Things (IoT) products. These wearable devices perform many everyday functions characterized as mobile information processing comprising of such functions like email, navigation, fitness tracking, health monitoring, and many more [1]. Due to a wide range of functionalities, the wearable market is expected to reach 21.4 billion dollars by 2024 [2] with an expected annual growth rate of 27%.

Unlike general-purpose systems, consumer wearable devices are expected to perform a wide range of tasks within stringent area and energy constraints. Despite these stringent design constraints, wearable devices are increasingly processing complex workloads due to increasing consumer demand. To meet these computational demands, it is essential that wearable devices feature computational resources that satisfy the workloads' requirements without introducing substantial overhead.

There have been some prior research efforts on enhancing the performance of wearable devices. For instance, Tan et.al [3] proposed a low-power many-core architecture for wearables in order to parallelize the wearable workloads. The authors used a 16-core architecture, called LOCUS, to achieve 1.71x speedup while reducing power consumption by 44.8% compared to a quad-core ARM Cortex A7 processor.

✉ Dhruv Gajaria
dhruvgajaria@email.arizona.edu

Tosiron Adegbija
tosiron@arizona.edu

[1] Department of Electrical & Computer Engineering, University of Arizona, Tucson, AZ, USA

However, analysis from Liu et.al [4] showed that the thread level parallelism (TLP) in wearable workloads is typically limited; most wearable workloads do not require more than 2-4 cores to fully exploit the available TLP. As a result, a many-core architecture may remain underutilized in an area and power-constrained environment, and leave optimization potential untapped. On the other hand, application-specific integrated circuits (ASICs) may provide optimization for a specific algorithm. But ASICs would typically result in underutilization of the area and resources, and inflexibility to compute a different algorithm than the one for which the ASIC was designed. Furthermore, given the vast array of wearable computing applications, it is unrealistic to design ASICs for every application, given the non-recurrent engineering (NRE) cost of ASIC designs. Therefore, it is essential to consider different architecture paradigms to address these gaps in the wearable computing state-of-the-art.

In this paper, we explore the use of *domain-specific architectures (DSA)* to provide a happy medium between the efficiency of ASICs and the flexibility of general-purpose architectures in wearable computing systems. Unlike general-purpose architectures, which are typically optimized for average case performance, domain-specific architectures are specialized with computational resources to optimize a particular domain of applications with similar execution characteristics [5]. Compared to ASICs, DSAs substantially improve utilization while enabling performance and energy benefits for a variety of applications. Prior works have designed DSAs to improve the performance and energy of different application domains, such as ECG based authentication [6] and neural networks [7, 8].

In this paper, for the first time to the best of our knowledge, we analyze the benefits of domain-specific architectures for wearable workloads. First, we perform a robust analysis of a set of wearable workloads to identify their computational characteristics and similarities (e.g., bottlenecks, available parallelism, read-write characteristics, memory and compute boundness, etc.). Based on the workload analysis, we explore a variety of architecture optimizations that collectively specialize the architectures for wearable workloads. Based on our analysis of different architecture alternatives and the workloads' characteristics, we provision the domain-specific architecture with a set of optimizations. These optimizations include a relaxed retention time STT-RAM cache [9], a low-power SIMD architecture, a low access latency buffer to speed up the computations for signal processing applications, and a prefetch architecture to optimize processing that involves non-contiguous memory locations. Furthermore, to embrace the diversity of wearable workloads, we explore various DSA designs (in the context of single and multi-core systems) for different workload requirements and analyze their performance, energy, and area tradeoffs.

We make the following key contributions in this paper:

- We analyze the execution characteristics of several wearable workloads to reveal their performance bottleneck and execution similarities.
- Informed by the workload analysis, we perform a design space exploration to select architecture design schemes that can satisfy the computational requirements of wearable workloads.
- We study and analyze the performance, energy, and area tradeoffs of the different proposed design schemes.
- We compare our DSA designs to a base ARM Cortex A7 processor and show that our DSA designs achieve average performance and energy improvements of 2.94x and 39.65%, respectively. Compared to prior work—a 16-core architecture (LOCUS)—our single-core DSA reduced the average energy by 38% and reduced the area by 14.98x, while trading off 22.47% in performance.
- We also explored a dual-core variant of our DSA, which reduced the performance tradeoff of the single-core DSA to 1.7% compared to LOCUS, at the cost of energy optimization.

The rest of this paper is structured as follows: Section 2 provides a brief background of prior related work and motivates the wearable workloads considered herein, Section 3 describes and analyzes the wearable workloads considered; Section 4 explores the design space of DSA components, based on the workload analysis. Thereafter, Section 5 presents four DSA design schemes, which successively build on each other to satisfy different workload requirements. To evaluate the proposed DSA design schemes, Section 6 describes our experimental setup, and Section 7 presents and analyzes the results in comparison to prior work.

2 Background and Prior Works

In this section, we first present some brief background on signal processing workloads in wearable devices. Thereafter, we briefly discuss prior works on optimizations for wearable computing, focusing on architecture optimizations, for brevity, and provide an overview of the background on domain-specific architectures.

2.1 Importance of the considered workloads in wearable devices.

We considered a range of wearable workloads, dominated by signal processing workloads, for the analysis presented herein. These workloads have wide-ranging uses in wearable devices, and as such, require computing resources to

efficiently process the signals. In what follows, we describe a few examples of signal processing workloads that are represented in the workloads considered herein, and some of their use-cases. For brevity, we only focus on example use-cases and not on the low-level algorithmic details of the different signal processing applications.

Convolution is a mathematical operation that blends one function into another. It is commonly used in various signal processing and analysis applications like ECG signal processing [10], or even for human activity monitoring [11]. *Histogram* is an approximate representation of distribution of the data. Histogram is used in various applications like cognitive signal processing [12], hand gesture recognition [13], etc.

Discrete time warping (DTW), which measures the similarity between two temporal sequences, is a widely used signal processing technique. It has a wide range of applications, like motion tracking [14], activity recognition [15], gait recognition [16], etc. *Discrete wavelet (DWT)* transform (or Haar transform) can be interpreted as spectral analysis using a set of basic functions that are localized in both time and frequency [17]. It has a wide range of applications from signal filtering [15] to ECG authentication [18]. Apart from these signal processing applications, we also consider other applications that are commonly featured in wearable workloads, including *encryption*, *authentication*, *navigation*, and *neural networks*. Further analysis of our workloads is in Section 3.1.

2.2 Optimizations for Wearable Computing

Due to the stringent resource constraints of wearable devices, most prior works have focused on optimizing the power consumption of wearable computing. Proposed optimizations include using energy harvesting techniques, designing low-power ASICs or accelerators for specific applications, and optimizing general-purpose multi-/many-core architectures for wearable applications that exhibit high parallelism. For instance, prior works [19–21] proposed low-power hardware for healthcare and electrocardiogram (ECG) monitoring. The proposed work involved developing low-power wireless capacitive ECG sensors for wearables [19], and a system-level architecture of ultra low-power wireless body sensor nodes for real-time monitoring [20]. Dieffenderfer et. al. [21] explored energy harvesting for wearable devices using thermal radiation of motion of the body to power a multimodal sensing platform.

A common thread among prior work on architecture optimizations for wearable computing involves attempting to exploit as much parallelism as possible in wearable workloads. For example, Dogan et.al. [22] proposed a multicore solution for low-power healthcare monitoring systems. The system comprises of 8-core processors, shared multi-banked data and instruction memories, and flexible crossbar interconnects to leverage the parallelism in the workloads. Their work achieved power savings of 39.5% compared to a 32-bit ultra low-power reduced energy instruction set computer (ReISC) microprocessor [23].

Similarly, Tan et.al [3] proposed a low-power 16-core architecture (LOCUS) to improve the performance and power consumption in smartwatches. Compared to a quad-core Cortex A7 ARM processor, LOCUS achieved a 71% performance improvement while reducing the power consumption by 44.8% across all the kernels. In this work, we use LOCUS to represent the state-of-the-art against which we compare the domain-specific architectures proposed herein. While LOCUS achieved performance and energy benefits using parallelism in low-power cores, employing a many-core architecture would still result in high area overhead and under-utilization of resources. These overheads can be prohibitive for wearable devices. In this work, we explore DSAs to optimize wearable workloads for higher resource utilization and to achieve improved performance and energy consumption.

2.3 Prior Work on Domain-specific Architectures

Unlike an ASIC (otherwise referred to as accelerator), a domain-specific architecture focuses on optimizing a set of applications based on their similar execution characteristics. Many works have proposed domain-specific architectures for various application domains. For example, Google proposed Tensor Processing Units (TPU) to achieve performance per Watt improvements of up to 50x for inference in machine learning [24]. The TPUs employ 256x256 systolic array of multipliers, use narrower data, and eliminate unnecessary general purpose architecture features that may be critical for general-purpose CPUs. The goal was to increase performance and energy benefits by emphasizing the resources that are used by inference computations.

Cong et.al [25] proposed a domain-specific processor for 3D integration medical image processing. The proposed work utilizes 3D technology, for high memory bandwidth and low-latency accesses, to stack their multiprocessor and accelerators. The processor achieved 7.4x speedup and 18.8x energy savings compared to a base CPU for various applications in medical image processing. Tucci et.al [26] proposed a domain-specific architecture based on systolic arrays for DNA sequence alignment. Their work achieved 350x performance benefits and 790x power efficiency over an Intel Xeon processor. Similarly, domain-specific architectures have been proposed for post-quantum cryptography [27], accelerating sparse matrix multiplication [28], energy-efficient communication in IoT [29], and for data fusion based on Kalman filtering [30]. More recently,

Cordeiro et. al. [6] proposed a domain-specific architecture for secure, fast and energy-efficient ECG authentication. They proposed an architecture that provided constant timing across all the steps of the authentication application, in order to mitigate timing attacks while achieving 19% and 4.62x performance and energy benefits compared to an ARM Cortex A15 processor.

To the best of our knowledge, ours is the first exploration of domain-specific architectures for wearable computing. In order to design efficient DSAs and maximize their benefits, we must first understand the execution similarities that are featured in the domain. This is more so important for wearable devices, where resource constraints necessitate highly accurate specialization to wearable workloads' resource needs, without wasting area or power on unnecessary components. Furthermore, since wearable devices do not have the luxury of employing accelerators for individual applications or large many-core general-purpose processors, it is essential to explore architectures that optimize a set of applications resulting in high resource utilization, high performance, and low energy consumption, with minimal area overheads. To this end, we employ domain-specific architectures to gain performance and energy benefits, while minimizing the area overheads incurred by prior work.

3 Wearable Devices Workload Analysis

To efficiently design domain-specific architectures for wearable devices, it is important to first identify and understand the domain characteristics and execution requirements that are similar across the different workloads within the domain. Our goal was to design architectures that exploit the inherent characteristics of the workloads, especially the available parallelism, while avoiding substantial overheads to the system. Furthermore, we sought ways to reduce data movement, which is a common cause of bottlenecks, and opportunities for specializing the data precision to the workloads' requirements.

This section describes our analysis of wearable device workloads and the insights derived therein. We first enumerate the workloads used in our experiments. Thereafter, we present our analysis of the available parallelism in the different workloads, the similarities in the kinds of parallelism, and some of the challenges in parallelizing the workloads within the context of wearable devices' typical resource constraints.

3.1 Workloads

Since there is no existing unified benchmark suite for wearable devices, we used a variety of workloads from different sources. The majority of workloads were taken from [3], including kernels for *2D convolution, dynamic time warping (DTW), haar transform or discrete wavelet transform (DWT), histogram, AES encryption and decryption*, and *Astar*. Furthermore, we used the ECG biometric authentication workload used in [6], and implemented a kernel to implement multiply and accumulate (MAC) operations, to represent the most common function in neural network applications. The workloads, their domains, and input sizes are depicted in Table 1.

Table 1 Wearable workloads.

Category	Kernel	Input size
Image processing	2D convolution	300*300
Image processing	Histogram	300*300
Pattern Matching	Dynamic Time Warping	300*300
Compression	Haar Transform	300*300
Biosignal authentication	ECG	7500 samples
Navigation	Astar	3770 nodes
Encryption	Aes Encrypt	20 bytes
Encryption	Aes decrypt	20 bytes
Machine learning	Multiply Accumulate(MAC)	300*300

3.2 Workload Analysis

Our first step in the workload analysis was to identify the bottlenecks in the different workloads and, importantly, the similarities across the workloads. For our experiments, we used Intel Vtune Profiler [31]. Intel Vtune Profiler is a performance analysis tool that enables the static analysis of workloads. Intel Vtune can help developers determine how much time was spent in each portion of an application [32] in order to determine the bottlenecks present in the applications. To properly analyze the computationally relevant regions of the workloads, we skipped the initialization phase of the workloads. This phase consists mainly of fetching the data inputs from storage and initializing them into appropriate data structures. The computing architecture has minimal impact on this phase and as such should not be an optimization focus.

One of the first observations while analyzing the different workloads is that there was typically a single function or loop that consumed a majority of the computational time during the execution. This observation is illustrated in Figure 1, which depicts the ratio of the biggest performance hotspot to the overall computational time for the workloads. As seen in the figure, in the smallest case, a single loop accounted for 78% of *astar*. For *haar* and *histogram*, a single loop accounted for their entire computations,

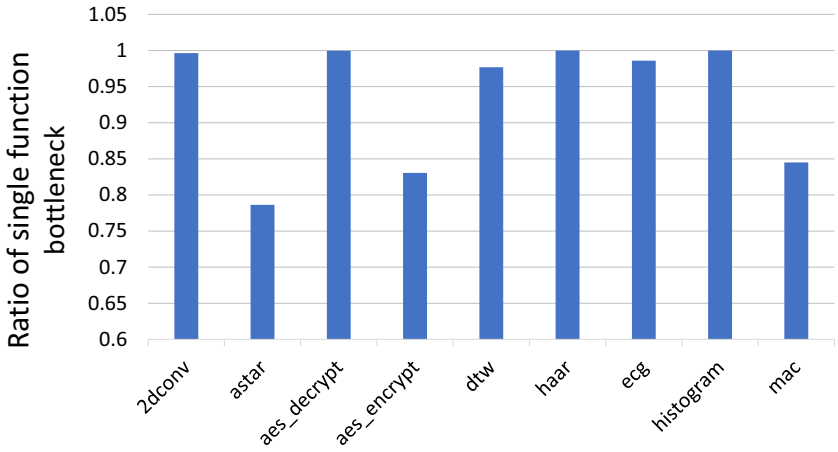

Figure 1 Ratio of largest compute-intensive loop or function to the overall computation time.

with the other workloads exhibiting similar bottleneck characteristics.

Given the prominence of loops in the different workloads' bottlenecks, we further analyzed the loops to identify opportunities for exploiting the inherent parallelism within the loops. Specifically, we explored the potential amenability to, and benefits of, *vectorizing* the different loops, including the similarities in the vectorizability of the different workloads' loops. *Vectorization* is an optimization technique that exploits the data-level parallelism in an application to simultaneously perform the same operation on multiple data elements. Vectorizability in a program can be analyzed by the compiler and can occur over countable loops, functions, and basic blocks [33, 34]. A program's amenability to vectorization depends on the amount of data dependencies present in the program's loops. Some loops or code blocks are easily vectorizable due to the presence of few/no data dependencies, constant data types, or memory strides. However, some loops are not readily vectorizable due to data dependencies or irregular data accesses, and may require code restructuring or additional memory hardware to derive the benefits of vectorization [35, 36]. Figure 2 illustrates an easily vectorizable loop compared to a non-vectorizable loop. In the vectorizable loop, the different loop iterations are independent of each other, whereas in the non-vectorizable loop, current iterations depend on previous results of the loop.

We observed some similarities and differences in the vectorizability of the different workloads. For instance, kernels like *2dconv*, *dtw*, and *haar* had easily vectorizable loops due to the absence of data dependencies across the loop iterations. However, kernels like *histogram*, *aes_encrypt* and *aes_decrypt* had conditional dependencies in their loops, which made them less amenable to vectorization. Similarly, *astar*, which is a graph workload that accesses irregular but predictable memory locations, was difficult to vectorize. Finally, most of *ecg*'s execution time was spent in a single function (segmentation) that featured complex branching statements and multiple exit conditions, thereby making it hard to vectorize.

Apart from the parallelism observed in the different workloads, we also found that some of the applications were memory or cache bound. For this analysis, we used the GEM5 simulator [37] to model a Cortex A7 ARM processor and gathered the statistics on the workloads' memory activities (e.g., memory references, cache miss rates, average memory access latencies, etc.) to identify the memory bottlenecks. We observed that workloads like *2dconv*, *dtw*, *astar*, *haar* and *histogram* exhibited high amounts of data movement, and in effect, had high memory bottlenecks compared to the other applications. Similarly, *mac* exhibited a high data bottleneck, although, to a lesser extent than the aforementioned workloads. We also found that for all the workloads, except for *astar*, the main source of memory activity resulted from data on which the algorithms were performing computations. As a result, majority of the stress was on the data caches. *Astar*, on the other hand, also exhibited high memory activity in

Figure 2 Examples of vectorizable and non vectorizable loops.

```
foo()
{
  int i;
  for ( i = 0; i < 256; i++ )
    {
      a[ i ] = b[ i ] + c[ i ];
    }
}
foo.c:4:4: note: LOOP VECTORIZED
```

```
sfoo()
{
  int i;
  for ( i = 0; i < 256; i++ )
    {
      a[ i ] = a[ i - 1 ] + a[ i ] + a[ i + 1 ];
    }
}
sfoo.c:4:4: note: LOOP NOT VECTORIZED
```

the instruction cache, which had high amounts of conflict misses due to the low reuse of the workload's instructions. We also observed that all the wearable workloads were read-intensive, and exhibited high read-to-write ratios, ranging from 2.3 for *aes_encrypt* to 17.11 to *2dconv*.

In what follows, we describe how the analyses presented in this section informed the architecture design decisions we made for designing the domain-specific architecture.

4 Domain-Specific Architecture Design Space Exploration

Given the general characteristics of the wearable workloads, we explored three different optimizations for the domain-specific architecture: *SIMD computing* for the highly vectorizable workload portions, *incorporating a buffer* to reduce memory bottlenecks, and *prefetching* to preemptively access predictable but non-contiguous memory locations. Our exploration process followed a simple heuristic as depicted in Figure 3. The figure describes our exploration process for selecting optimizations for the domain-specific architecture

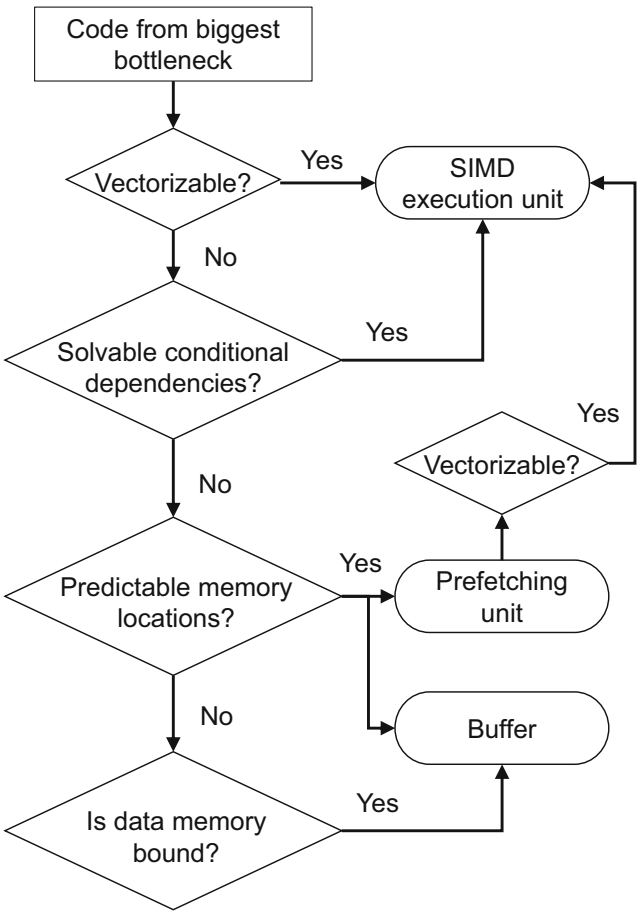

Figure 3 Flowchart representing a high-level overview of our exploration process.

based on the workload analysis. We performed the process for each of the workloads considered in this work.

To satisfy the majority of wearable workloads, which featured large data-parallel loops, we opted to feature low-power single-instruction multiple-data (SIMD) [38, 39] execution units within the processor core to parallelize the computations. We chose to use SIMD units as opposed to multi-threaded execution as in prior work [3]. Multi-threaded execution would lead to initialization overheads, data partitioning issues, or software interrupts and overheads from other cores which could be prohibitive for wearable workloads. These workloads typically have fewer instructions, and thus, they are less likely to derive high benefits from multi-threading [40, 41]. Furthermore, even though some of the workloads, like *histogram*, *aes_encrypt* and *aes_decrypt* were not readily vectorizable, we found that their codes could easily be modified and restructured to derive SIMD benefits. The modifications involved utilizing more memory space to rearrange the data to eliminate conditional dependencies.

Some applications like *astar*, similar to other graph applications, do not access contiguous memory locations, thus making them more difficult to vectorize. However, we found that the memory locations used in the graphs were predictable and thus could be prefetched and stored into an array of contiguous locations to derive SIMD benefits. To enable efficient prefetching, we designed a simple prefetcher and also incorporated a buffer, implemented as a tightly-coupled memory, to store the prefetched data in order to reduce cache misses. Our prefetcher design is described in Section 4.3.

To further reduce the overhead, given the tight resource constraints of wearable devices, we employed a low-power STT-RAM buffer to minimize the cost of data transfer. STT-RAM has several qualities that make it a viable option for resource-constrained wearable devices. Most importantly, STT-RAMs cells have near zero leakage power and require about 1/9 to 1/3 area of SRAMs cells. However, STT-RAMs also suffer from long write latency and energy [42] and are therefore more suited for read-intensive workloads. Prior works have shown that the write latency and energy overheads can be mitigated by relaxing the non-volatility of STT-RAM, resulting in a *relaxed retention STT-RAM* [43]. In our analysis of wearable workloads, we found that the workloads exhibited low data reuse and also featured short-lived data blocks. As such, memories do not need to retain data for long periods of time, thus making the relaxed retention STT-RAM a suitable option for these workloads. Based on our analysis, we found that a $75\mu s$ retention time sufficed for the wearable workloads. While some relaxed retention STT-RAM designs have featured refresh mechanisms to maintain data integrity after the retention time has elapsed [42], we found this mechanism

to be unnecessary and found that allowing memory blocks to expire did not negatively affect the performance of the wearable workloads. To prevent data from becoming unstable if the retention time elapses before a data block is evicted, we incorporated a low-overhead 2-bit monitor counter, similar to prior work [9], to proactively invalidate memory blocks before the retention time elapses.

In addition, STT-RAM is especially suited for wearable workloads due to the disproportionately high number of reads vs. writes. As such, even though STT-RAM typically has long write latencies (compared to SRAM) [42], the overheads are mitigated since majority of memory accesses in wearable workloads are read accesses. Thus, we also used STT-RAMs instead of SRAMs for our cache memories. Similar to prior work, we based the domain-specific architecture designed herein on an in-order ARM processor, similar to the ARM Cortex A7, as it is widely used in fitness and smart wearable devices [44]. We found out-of-order execution to be over-provisioned for the wearable workloads due to the limited amounts of complex branching and the hardware overheads of out-of-order execution.

In what follows, we describe in detail our implementation of the SIMD unit, the buffer, and the prefetcher.

4.1 SIMD Architecture for Wearable Computing

There are several possible implementations of SIMD architectures [45]. For example, SIMD units in Intel AVX [46] architecture allow the amount of vectorization to be specified based on the data type used. The architecture uses 256-bit vector registers, and a *long* datatype, which requires 64 bits to store one variable, thus allowing four simultaneous computations (i.e., a *vectorization factor* of 4). A *float* datatype would allow a vectorization factor of 8. Similarly, ARM NEON [47] uses 128-bit vector registers supporting four 32-bit data computations, eight 16-bit computations, or sixteen 8-bit computations. However, traditional SIMD architectures contain significant area and power overheads that would be prohibitive for wearable devices [48, 49]. Thus, we describe a low-power SIMD architecture that is more suitable for wearable devices.

First, rather than enable variable data types and vectorization factors as in traditional SIMD architectures, we decided to limit the data types and vectorization factors that could be executed on our SIMD execution units. To this end, we analyzed the workloads to identify their appropriate data precisions, the ideal vectorization factor, and what

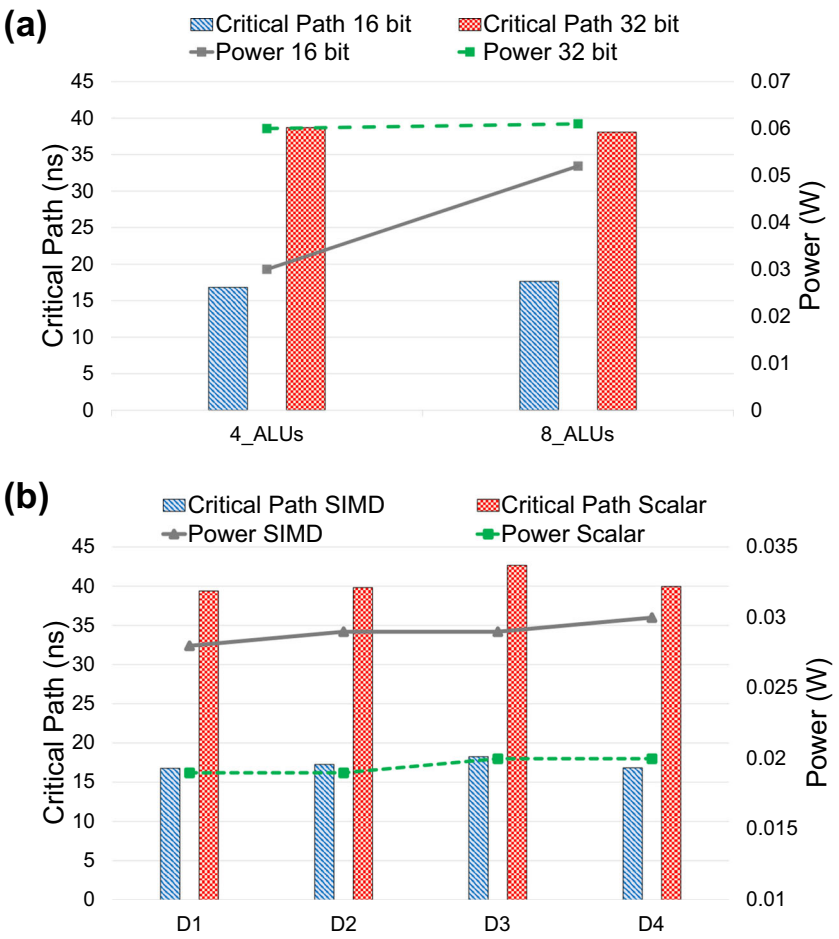

Figure 4 Critical path and power impacts of (a) different number and sizes of SIMD ALUs and (b) the number of supported ALU computations.

kind of ALU operations are required. Our evaluation metric for this exploration was the *power consumption*, *critical path*, and *resource usage* of the SIMD architectures. We performed RTL analysis of the the different design alternatives using synthesizable Verilog and evaluated them using Xilinx Vivado synthesis.

Figure 4a presents our analysis of vectorization factors of 4 and 8 and 16-bit and 32-bit SIMD ALUs. We limit the analysis to these numbers of ALUs since increasing them showed diminishing returns in optimization benefits. Analysis of the wearable workload data also showed that the 32-bit precision was overprovisioned for the workloads, and the workloads could be effectively run at 16-bit. This observation is in line with prior work [50–52]. We also analyzed the impacts of 16-bit and 32-bit ALUs on the critical path. The results showed that while the critical path was not significantly impacted by a change in the number of SIMD ALUs, the critical path significantly increased when the ALU was changed from 16- to 32-bit. Increasing the number of ALUs from 4 to 8 only increased the critical path by 4.75%, whereas the power increased substantially by 73%. Furthermore, increasing the size of the ALUs from 16 to 32 bits increased both the critical path and power consumption by 2x. We also found that increasing the number of ALUs increased the resource usage by 45%, while increasing the ALU size from 16 to 32 bits resulted in a 2x increase in resource usage. Given these analyses, we opted to use four 16-bit SIMD ALUs to enable at least four simultaneous computations.

Secondly, we observed that the computations in the wearable workloads were dominated by ALU operations. Different workloads like *2dconv*, *ecg* mainly had ADD, SUBTRACT, and MULTIPLY operations, *histogram* and *haar* mainly comprised of DIVIDE, SUBTRACT, and ADD operations. Workloads like *aes_encrypt* and *aes_decrypt* mainly had XOR, SHIFT, and ROTATE operations; *dtw* also had AND, OR, and COMPARE operations, while *mac* mainly had multiply-accumulate operations. Thus, we explored the appropriate kinds of ALU resources required and the impacts of the ALU operations on power, critical path, and resource usage. Based on the workload analysis, we explored four ALU design alternatives, which we call, for simplicity, *D1, D2, D3,* and *D4*. *D1* has basic ALU operations of ADD, SUBTRACT, MULTIPLY, and DIVIDE; *D2* has operations in D1 plus XOR, ROTATE and SHIFT; *D3* has D2 plus all the compare operations (e.g.,greater than, less than, greater than zero or less than zero). *D4* has D3 plus multiply-add operation which dominates neural network applications.

Figure 4b presents the impact of the different design alternatives on critical path and power. For this experiment, we used an ALU of 16-bit data. As seen in the figure, increasing the number of ALU operations did not have a significant impact on the critical path and power. Compared to D1, the critical path increased by 2.8%, 8.8% and 2.85% for D2, D3 and D4, respectively, and the power increased by 3.5%, 5.3%, and 7.14% respectively. We also evaluated the resource usage due to the addition of ALU operations and found that compared to D1, the resource usage increased by 78.35%, 88.76%, and 92.51%. Despite the increased resource usage compared to D1, we opted to use D4, since it includes all the required ALU operations to cover as many wearable workloads as possible. We also compared the SIMD ALU units with the base scalar ALU which supports 32-bit data and found that the critical path of the SIMD ALU was 0.42x that of the scalar ALU. However, the SIMD ALUs' power was found to be 1.52x the power of scalar ALU due to more computational units, demonstrating some of the tradeoffs in domain-specific computing.

4.2 Buffer to Reduce Data Movement

Our main goal in exploring the use of a buffer was to reduce the time and energy costs of data movement in data-rich applications. In general, signal processing workloads that process data from wearable sensors exhibit higher data movement, as exemplified by the *ecg* application. To save memory requirements, this application stores a part of the ECG signal, performs some computations on the data, and then combines the newer incoming signal with the analyzed results, as illustrated in Figure 5. This results in a lot of extra computations, and can substantially increase the computation time and energy consumption due to the data movement. To mitigate this overhead, we used a tightly-coupled low latency buffer that allows the processor to directly access stored raw signal samples, thereby enabling the processor to analyze a larger portion of the signal. This buffer significantly reduces the number of computations and data movement by up to 89% compared to the system without a buffer. Even though we incorporated the buffer specifically for the *ecg* workload, the buffer can also be utilized by other applications that process raw signals samples, e.g., sleep detection, SpO2 analysis, exercise tracker, etc [53, 54].

```
Segmentation() {
  int i;
  int j;
  for (i = 0; i < SIGNAL_LENGTH; i++) {
     if (SIGNAL_LENGTH > BUFFER_SIZE) {
       for (j = 0; j < BUFFER_SIZE; j++) {
           BUFFER_SIGNAL[ j ] = BUFFER_SIGNAL[ j + 1 ];
       }
     }
      PeakDetection();
  }
}
```

Figure 5 Example of ECG code which shows dependency on buffer size.

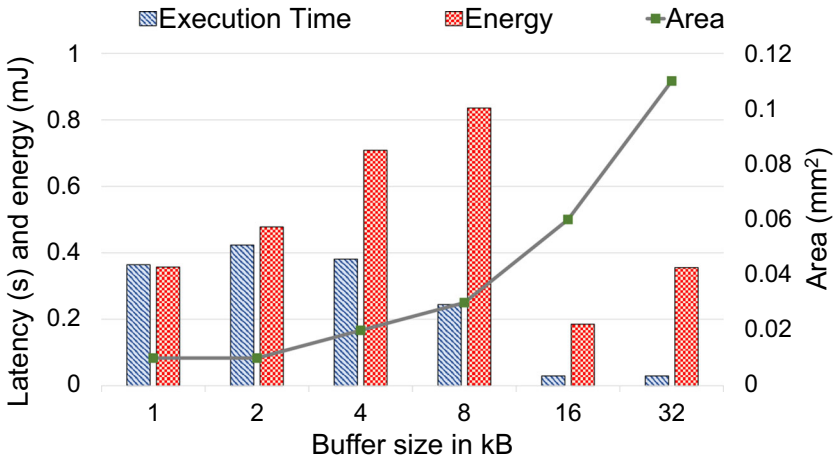

Figure 6 Impact of buffer size on ECG authentication execution time in seconds, buffer energy in mJ and buffer area in mm^2.

To determine the appropriate buffer size, we explored a variety of sizes to determine their impacts on the execution time and the energy/area tradeoffs. Figure 6 depicts the execution time for different buffer sizes while running the *ecg* workload. As seen in the figure, as the buffer size was increased, the execution time of the *ecg* application reduced significantly. There was a small increase in execution time when going from 1kB to 2kB buffer size because the whole signal did not fit in the buffer; as such, more computations were required to compare the current signal with the previously obtained computations. We observed that a 16kB buffer was sufficient to store the whole signal using a 16-bit data precision; further increases in buffer size did not yield any significant benefits. Thus, we used a 16kB buffer for our architecture. Furthermore, the buffer accesses required by the *ecg* application were mainly to contiguous memory locations, and no write operations (i.e., store instructions) were made to the buffer from the CPU. As a result, the *ecg* workload had a high read-to-write ratio to the buffer. Therefore, we used STT-RAM for the buffer to reduce power and area overheads.

4.3 Prefetch Unit

The use of a prefetch unit was inspired by the fact that some of the wearable workloads were not easily vectorizable, not because of data dependencies, but because the memory addresses were non-contiguous. Thus, we decided to incorporate a low-overhead prefetch unit to preemptively fetch non-contiguous but predictable memory addresses and store them into contiguous memory locations for easy vectorization—this process is analogous to the gather operation in vector architectures. Given the increasing popularity of graph kernels in wearable devices [3, 55], our goal was to design a domain-specific architecture that could improve the computation of graph kernels' general characteristics. Graph kernels have a structure comprising of nodes and edges, wherein each of the nodes has its own specific data

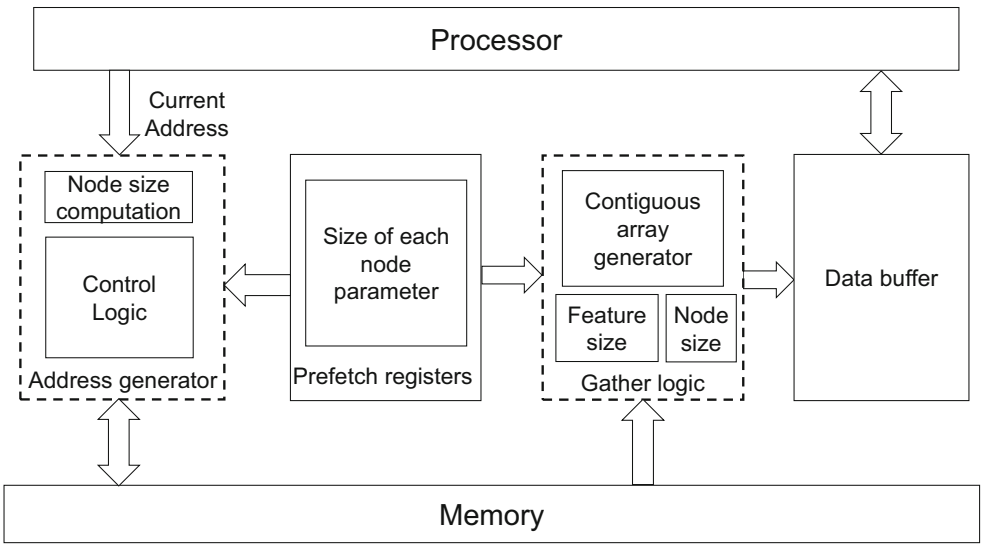

Figure 7 Prefetch unit architecture.

or information with fixed data sizes, and each parent node can point to one or multiple child nodes. By keeping track of the number of successive nodes, it is possible to get all the required node parameters of successive nodes and arrange them into contiguous arrays. Thereafter, vectorization can be used to perform simultaneous computations on the data of all the successive nodes.

One approach to achieve this vectorization is to compute the addresses of the successive nodes, store them in an array, and use a gather architecture [56] to arrange them in contiguous memory locations for vectorization. Intel's SSE [57], AVX [46] architecture supports these kinds of operations. However, considerable CPU time is wasted in computing the successive node addresses even though they have predictable structures. In our work, we explored low-overhead architectures that help compute the successive node addresses, thus freeing up the CPU for other computations. Hence, the successive node parameters can be fetched and arranged in contiguous memory locations without the CPU's intervention.

Figure 7 depicts the architecture of our prefetch unit. The prefetching architecture comprises of three main parts: *address generator*, *prefetch registers*, and *gather logic*. The prefetch registers hold the data size and offset addresses of the node data to be searched once the current node address is known. The address generator generates the addresses of data to be prefetched from the successive nodes. The gather logic arranges the data of the successive nodes coming from the memory into contiguous memory locations. The data to be stored in prefetch registers depend on the node structure and definition, and thus are known at compile time. In the address generator, the address of the current node is passed to the node size compute block, which computes the current node's number of successors. To efficiently utilize this unit, we modified the program code to ensure that the node characteristics remain fixed at specific location offsets from the base node address, which can be stored in the prefetch register. After accessing this information, the address generator can obtain the size of the node and the successive node addresses from the memory and store it in prefetch registers and gather logic. Once the successive node's base address is determined, the address of the other associated data (e.g., distance from destination in a path finding algorithm) can then be computed since the data types and node structure remain unchanged from compile time. Addresses are generated and sent to the memory. We opted to directly connect the prefetch unit to the memory and bypass the cache due to the potential for high cache miss rates and low node reuse in graph traversal applications [58].

After the addresses are generated and return successive node features, they are arranged into a contiguous array using the gather logic. For example, for *astar*, the predetermined distance from the destination data for all successive nodes can be stored in one array in contiguous memory locations. This is possible because the gather logic contains information on node size, i.e., number of successive nodes found using node size computation and size of data required for each node parameter which is known at compile time. Similarly, our prefetch unit can work for other the graph traversal kernels that have predictable graph traversal structures (e.g., *breadth first search*).

5 Domain-Specific Architecture Design Schemes

This section describes different design schemes for our domain-specific architectures based on the analysis in Section 3 and the components described in Section 4. We present three design schemes herein, with each successive architecture building on the previous one to improve the optimization potential. For each of our designs, we also present the software modifications or functions needed to enable proper utilization.

5.1 DSA_vect

As seen earlier, the critical path of the SIMD ALU is 0.42x the critical path of the scalar unit. Thus, for our execution design, we implemented both scalar and SIMD ALUs to support different kinds of wearable workloads. The critical path of the execution units is bounded by the scalar ALUs, with the SIMD ALUs completing their computations within 42% of the clock period. To take advantage of the clock period to enable a high vectorization factor, while also limiting the area overhead, we designed the SIMD ALU to run eight computations in one cycle using only four parallel execution units. We achieved this by running the first four elements in the first half of the cycle and the next four in the second half of the cycle. Figure 8 depicts our SIMD architecture design. As seen in the figure, the execution unit comprises of a 16-bit SIMD ALU and a 32-bit scalar ALU. We used a 128-bit vector register to store eight 16-bit data elements to be computed in the SIMD units. We also incorporated flip-flop based dual-edge selectors that select the top four data elements at the positive clock edge and the next four at the negative clock edge. Implementing the flip flop-based selectors only increased the latency of the SIMD ALU increased by 15%, resulting in a 48% utilization of the critical path by the SIMD ALU. As such, the critical path remained bounded by the scalar ALU. To further enable area and power savings, we replaced SRAM caches with STT-RAM caches for all our designs due to high read-write ratio of wearable workloads as seen in Section 3.2.

On the software side, to efficiently utilize the proposed architecture, the wearable applications must be implemented using 16-bit data types (*int_16* and *float_16*) to

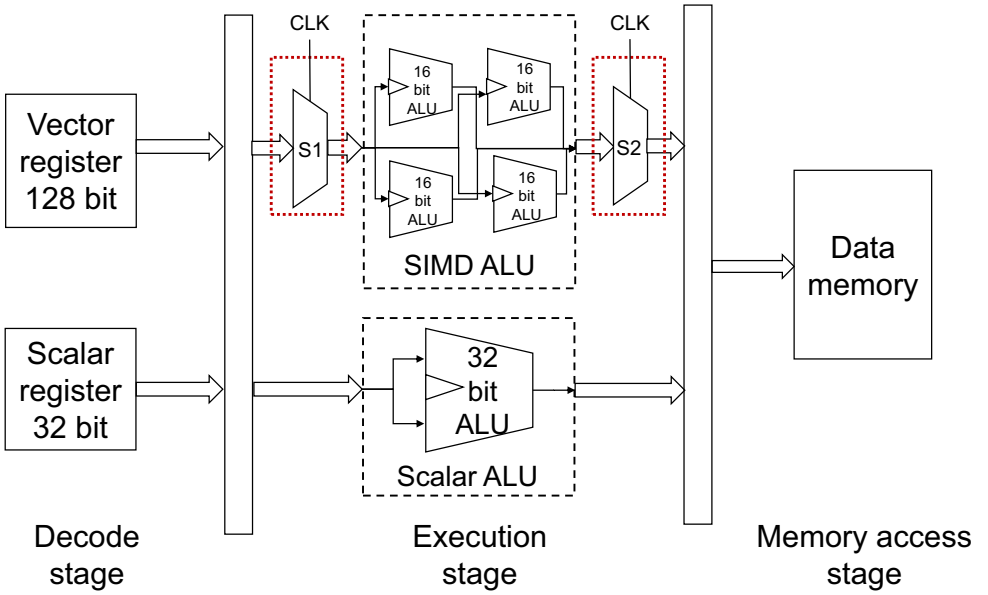

Figure 8 High-level overview of our SIMD architecture representation.

perform the 16-bit computations. The applications can be compiled using ARM GCC compilers, which support SIMD operations and are supported by the proposed SIMD architecture. As such, wearable application developers can develop applications to efficiently run on our proposed architecture with minimal effort.

5.2 DSA_buff

DSA_buff builds on *DSA_vect* by incorporating a tightly-coupled memory buffer to enable data accesses without the unpredictability of caches. Figure 9 illustrates the integration of the buffer into our architecture. The data in the buffer can be transferred directly from the main memory or the wearable sensors. The data can then be accessed directly by the processor via a multiplexer. The processor sends out a select signal *CS* to select the incoming data from the cache or buffer and another select signal *SS* to determine whether the data from wearable sensors goes directly into the buffer or to the main memory. To avoid data consistency issues, we only allow CPU load operations from the buffer. Furthermore, since the buffer operations are dominated by reads, we use an STT-RAM buffer to reduce the leakage and area overheads.

To enable the buffer utilization for wearable workloads, we implement three high-level specialized functions for buffer management called *buffAlloc*, *buffStore*, and *buffFree*. *BuffAlloc* allocates memory addresses to be utilized by the buffer and *buffFree* frees the memory addresses for use by other applications. Since wearable applications are typically known a priori, the programmer can allocate an application's signal data size for the buffer in order to prevent those address spaces from being simultaneous used by other applications. The function *buffStore* stores the data in the buffer, and this data will be accessed by the processor only through the buffer.

5.3 DSA_prefetch

Figure 10 illustrates the integration of the prefetcher with our architecture. The base structure of the prefetcher is as described in Section 4.3. This architecture features the prefetcher along with a prefetch buffer to store the

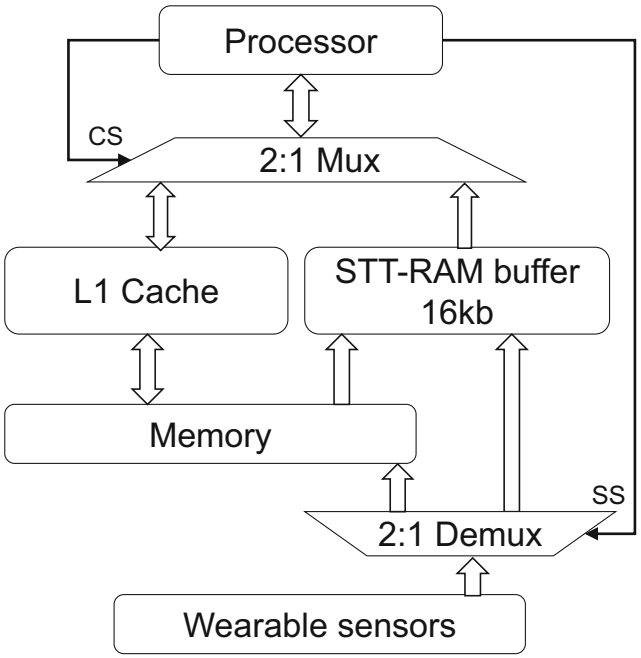

Figure 9 System architecture illustrating the buffer integration.

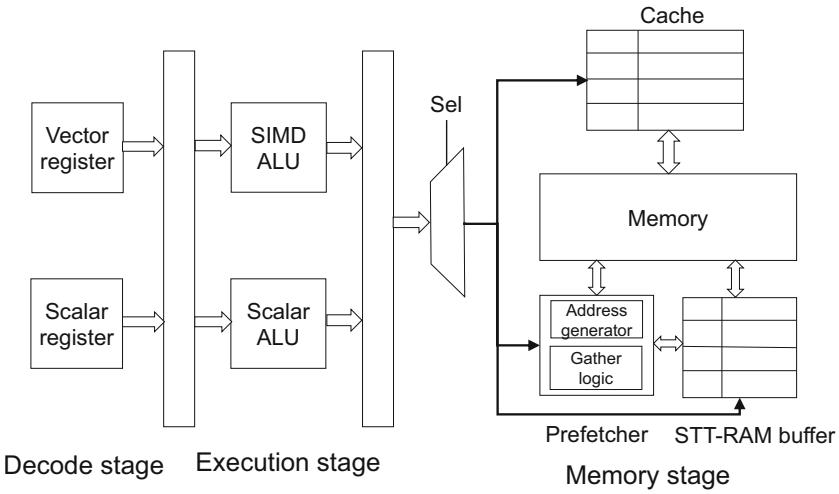

Figure 10 High-level illustration of prefetcher integration in *DSA_prefetch*.

prefetched data from the memory. As seen in the figure, the buffer and the prefetching architecture are completely independent of the cache and the processor needs to send an instruction that specifies if the cache is to be accessed or not. Note that the need for a prefetcher is not unique to wearable workloads. The *DSA_prefetch* design can be employed in other domains as well to help prefetch data for vectorization. However, for wearable devices, the buffer has broader use since it can be used by other applications like ecg, sleep detection, and graph applications via the prefetcher. In general-purpose workloads, buffer utilization will happen mainly via the prefetcher for graph workloads. Thus the *DSA_prefetch* design will have a higher utilization for wearable devices.

To integrate this architecture into wearable devices, we use the same buffer functions described in Section 5.2. For the prefetcher operations, the programmer needs to specify whether or not the graph will use the prefetcher and the needed prefetching parameters, to avoid unnecessary prefetching. To this end, we propose a *prefetch:init(graph, parameters_to_prefetch)* function, which enables the programmer to specify these parameters. During the application initialization, different node parameters will be loaded into the prefetcher registers. When the address of the successive node is generated and needs to the prefetched, a function *prefetch_successors(address)* can be called to pass the generated address to the prefetcher, thus arranging the parameters of all the successive nodes into contiguous memory locations within the buffer.

5.4 DSA_energy

The previous architectures assume a maximum utilization of necessary resources to optimize performance for the wearable workloads. However, given the stringent energy constraints of wearable devices, utilizing all available resources may not be suitable for energy optimization. For example, some workloads may not need the buffer in *DSA_buff*, thus wasting energy while running those workloads. To mitigate these overheads, we explored the *DSA_energy* architecture, which aims to increase energy savings compared to the prior architectures. The overall design of *DSA_energy* is same as *DSA_prefetch*. While previous design schemes use all the available hardware resources, the goal of *DSA_energy* is to only use additional hardware when necessary; if unused, the additional hardware is power-gated to save energy. Prior work [59] has shown that power-gating can be achieved in resource-constrained systems with minimal performance and area overheads. To help developers adapt the architecture to the running applications, a performance analysis and estimator tool like ARM Streamline Performance Analyzer [60] can be used to provide insights, through a priori analysis or through runtime hardware performance counter monitoring, about benefits of the different design schemes for latency and energy optimization on a per-application basis. We note that the development of the performance estimation tool is outside the scope of this paper.

6 Experimental Setup

In this section, we describe our experimental setup for analyzing the different domain-specific architectures proposed herein. We analyzed the workloads using Intel Vtune to determine performance bottlenecks. We used C and C++ flags along with the G++ compiler to validate that the functions were vectorizable. We modified some of the workloads as described in Section 3.2 to improve their vectorizability. To estimate the optimization potential from vectorization, we converted the programs into assembly instructions and calculated the in-loop scalar and in-loop vector computation costs for each loop. The scalar costs

of the loop include loop operations, function or class addressing, and stack operations. The vector costs include load and store operations, and various ALU operations as seen in the ARM or RISC-V vector ISAs [61, 62].

For our base processor, we used a single-core 22nm Cortex A-7 processor with a 1GHz clock speed and separate 32kB SRAM data and instruction caches, similar to many wearable devices [3]. To evaluate the behavior of relaxed retention L1 STT-RAM caches and buffer used in our DSA designs, we used an in-house modified GEM5 simulator that allows us to model the execution of both SRAM and relaxed retention STT-RAM caches. To model the energy and access latency of the caches, we used the NVSim [63] modeling tool and combined with execution statistics from the GEM5 simulator to determine the per-application cache energy and latency. Table 2 presents our SRAM and STT-RAM cache latency, energy, and area numbers. To estimate the processor area and power numbers, we used McPAT [64] and adjusted the power and area for STT-RAM accordingly. We synthesized our architecture designs in Xilinx Vivado [65] on a Kintex Ultrascale board and used McPAT to map the area and power impacts of our hardware designs to the base processor.

7 Results

This section presents the results of the performance and energy analysis of our single-core DSA designs in comparison to the base processor as well as to a dual-core ARM processor. We also compared our designs to prior work represented by *LOCUS* [3]. LOCUS is 16-core architecture that was designed for high-performance wearables and uses a Lightweight Message Passing protocol for inter-core communications. Next, we estimate the area and resource usage of the various design schemes, and finally, we present the power and area overheads. In total, this section compares seven different architectures: the four DSA designs described in Section 5, the base processor, dual-core ARM processor, and prior work (LOCUS).

Table 2 SRAM and STT-RAM cache parameters.

L1 cache configuration	32KB, 64B line size, 4-way	
Memory device	SRAM	STT-RAM
Retention times	–	75μs
Hit latency	0.486ns	0.445ns
Write latency	0.350ns	0.981ns
Read energy (per access)	0.0076nJ	0.003nJ
Write energy (per access)	0.0066nJ	0.035nJ
Leakage power	34.265mW	13.659mW
Area	0.023mm^2	0.012mm^2

To enable a fair comparison against the dual-core ARM processor, we assumed that the workloads running were optimized for multithreading. We estimated the multithreading operations by implementing the workloads using *OpenMP* [66], a popular multi-threading API that is compatible with both ARM and Intel frameworks. We annotated the code with the computational part the kernels to be optimized using OpenMP. This approach enabled us to use microarchitecture independent annotations rather than microarchitecture-specific hardware performance counters to approximate the performance optimization obtained through OpenMP. Furthermore, this approach enabled us to perform a stringent comparison of domain-specific architectures against the upper bound of achievable optimization using multicore/many-core systems. All the simulations were performed using a 22nm technology node.

7.1 Performance Analysis

Figure 11 presents the DSA performance results normalized to the base processor. Compared to the base processor, we observed average performance gains of 1.72x, 2.89x, 2.948x, 2.946x, 1.52x, and 3.32x for *DSA_vect*, *DSA_buff*, *DSA_prefetch*, *DSA_energy*, ARM_dual, and LOCUS, respectively. *DSA_vect* achieved significant performance gains for most of the workloads, except for *ecg* and *astar* because of non-vectorizable codes, compared to both the single and dual-core ARM processors. *DSA_vect* improved average performance by 13% compared to the dual-core ARM architecture while *DSA_buff* and *DSA_prefetch* improved the average performance by 1.9x and 1.93x respectively. We also observed that implementing a buffer for the *ecg* application improved the performance gains by 12.05x compared to *DSA_vect*. Similarly, we observed significant performance benefits for *astar* when we used the prefetcher. *DSA_prefetch* improved the performance for *astar* by 43.29% and 53% compared to the base processor and *DSA_vect*, respectively.

As mentioned earlier, we used *DSA_buff* in an attempt to reduce the memory bottlenecks for the different workloads. However, majority of the benefits of *DSA_buff* was observed for *ecg*, wherein the complete ECG signal was fit in the buffer during execution. As a result, the whole signal could be analyzed in one loop iteration, thereby substantially reducing the execution time for that workload. For all the other applications, *DSA_buff* only improved performance by about 0.5% compared to *DSA_vect*, but by 82% compared to the base processor. For *DSA_energy*, where the programmer can select the DSA units to optimize the energy consumption as described in Section 5.4, the performance deterioration was only 1% compared to *DSA_prefetch*, which was the best design scheme for performance optimization.

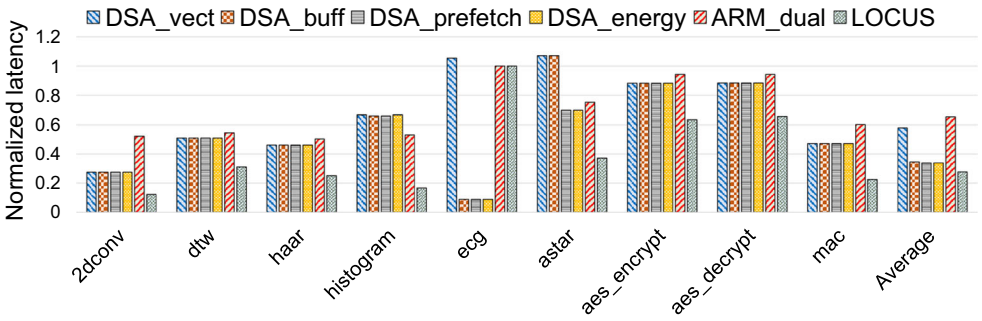

Figure 11 Latency of different architecture schemes normalized to the base.

For performance, LOCUS achieved the best results for all the workloads, except for *ecg*. On average, LOCUS outperformed the base processor by 3.61x and our best performing architecture, *DSA_prefetch*, by 22.47%. LOCUS outperformed our work with respect to performance due to a much higher number of parallel execution units. It is worth noting, however, that even though LOCUS had 16 cores, the performance improvement over our single-core *DSA_prefetch* arguably modest at 22.47%. LOCUS was unable to parallelize the *ecg* workload because the bottleneck function exhibited high data dependencies that were not amenable to parallelization.

We observed that replacing the SRAM with STT-RAM memories resulted in an average latency penalty of 7%. The latency degradation was as high as 11% for dtw, which exhibited higher write-to-read ratio and higher conflict misses than average. In general, the penalty of using STT-RAM caches over SRAM caches was modest for our workloads due to the high read-to-write ratio and higher cache block utilization with contiguous data computations. Despite the latency penalties of using STT-RAM, our work still achieved substantial improvements for the different workloads. For instance, for $histogram$, which required code modifications for vectorization, *DSA_vect* improved the performance by 49% via partial vectorization wherein only a part of the computations were vectorized due to conditional dependencies. In contrast, for $aes\_encrypt$ and $aes\_decrypt$, our work also achieved a 13% performance gain compared to the base processor due to small input size and fewer vectorizable elements in the workloads.

7.2 Energy Analysis

Figure 12 presents the DSA energy analysis for the different design schemes. Compared to the single-core ARM processor, *DSA_vect*, *DSA_buff*, *DSA_prefetch*, *DSA_energy*, and LOCUS reduced the energy by an average of 24.32%, 32.87%, 34.13%, 39.65%, and 16.45%, respectively, while ARM_dual *increased* the energy by 15%. Significant energy savings was achieved by all our design schemes for workloads like $2dconv$, dtw, $haar$, mac and $histogram$, which involved direct array computations. For $aes\_encrypt$ and $aes\_decrypt$, multithreading did not result in much latency optimization due to low input size and high initialization overheads. As a result, the muticore systems (ARM_dual and LOCUS) increased the energy consumption for $aes\_encrypt$ and $aes\_decrypt$ by 40% and 23%, respectively, compared to the base processor. Conversely, *DSA_energy* reduced the energy consumption for these workloads by 4.61% and 4.4%, respectively, compared to the base.

For all the workloads, *DSA_vect*, *DSA_buff*, and *DSA_prefetch* achieved higher energy improvement than ARM_dual. For $2dconv$, mac, and $histogram$, LOCUS achieved higher energy savings compared to our domain-specific architecture design schemes. We attribute this to the highly parallelizable nature of these workloads that gained huge performance benefits, which in effect, resulted in significant energy savings despite the power overheads of LOCUS. However, on average across all the applications, *DSA_energy* reduced the average energy by 38% compared to LOCUS. These results exemplify

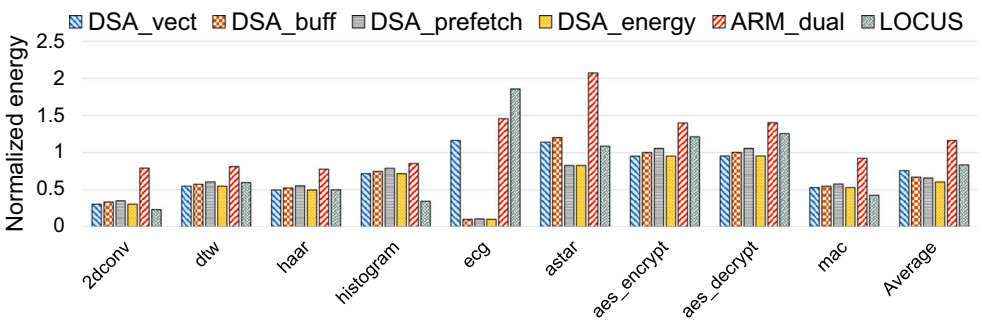

Figure 12 Energy of different architecture schemes normalized to the base.

the *DSA_energy* architecture's ability to adapt to different workloads' resource requirements in order to save energy.

We also note that replacing the SRAM cache with STT-RAM cache reduced the average energy consumption by 33.12% for the data cache and by 9.19% for the instruction cache. The STT-RAM caches reduced the overall processor average energy consumption by 15.13% with maximum energy reduction of 17.56% for *histogram*.

As previously alluded to, *DSA_buff* and *DSA_prefetch* keep all the available resources on while running all the workloads. However, not all applications were benefited significantly by using the prefetcher and buffer. As such, this over-provisioning increased the energy consumption of *DSA_buff* and *DSA_prefetch* compared to *DSA_vect* for some workloads. For instance, for $2dconv$, dtw, $haar$, $histogram$, $aes\_encrypt$, $aes\_decrypt$ and mac, *DSA_buff* and *DSA_prefetch* increased the average energy by 4% and 10%, respectively, compared to *DSA_vect*. Overall, *DSA_energy* achieved the highest average energy savings compared to all the design schemes and prior work due to the effective utilization of domain-specific architecture components, wherein unused components are dynamically power-gated to reduce the power consumption.

7.3 Extending the DSA to a Multicore System

The analyses presented so far have focused on the DSA design schemes implemented in a single-core system. We also explored and analyzed the impact of our proposed domain-specific architectures on multicore systems. For this analysis, we assumed that the workloads were broken down into different threads using OpenMP, where possible, and run on a system featuring two homogeneous *DSA_energy* cores. Within each core, the running thread used the domain-specific architectures, as required by the workloads, to optimize their computation. In this section, we compare a single core *DSA_energy* architecture and *M_dsa*—the dual-core *DSA_energy* architecture—to prior work, LOCUS, which is a 16-core architecture.

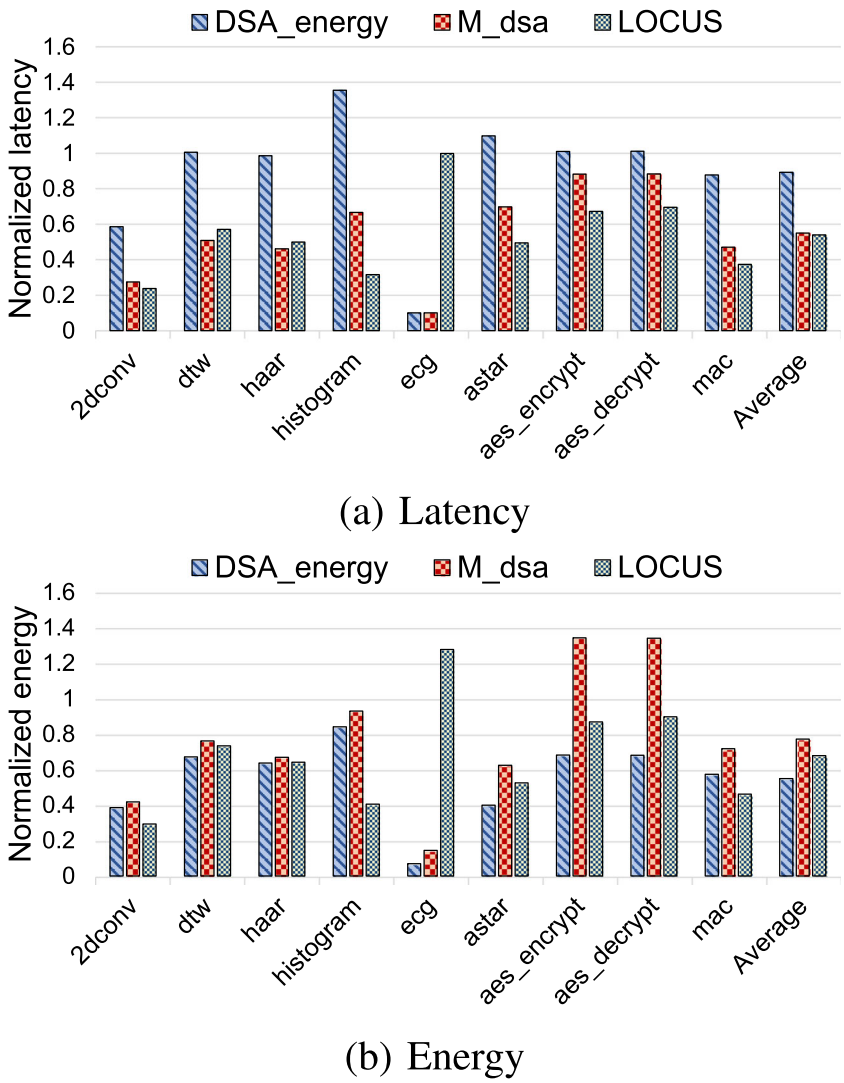

Figure 13 Latency and energy of single-core *DSA_energy*, dual-core *M_dsa*, and 16-core LOCUS normalized to the dual-core ARM architecture.

Figure 13a presents our experiments with multicore designs. From the figure, we observe that using the multicore architecture, *M_dsa* achieved performance benefits over the single core architecture across all the applications. Compared to the dual-core ARM processor, *DSA_energy*, *M_dsa*, and LOCUS improved the average latency by 10.69%, 45.01% and 45.97% respectively. We observed that *M_dsa* outperformed LOCUS for *dtw*, *haar* and *ecg* by 12%, 8%, and 9.91x, respectively. On average across all the workloads, the 16-core LOCUS only outperformed the dual-core *M_dsa* by 1.7%. These results illustrate the benefits of using a domain-specialized architecture vs. a general-purpose many-core architecture.

However, the energy results also illustrate the potential tradeoffs between latency and energy when using multicore processors in resource-constrained systems like wearable devices. As seen in Figure 13b, *DSA_energy*, *M_dsa* and LOCUS reduced the average energy by 48.20%, 21.9% and 28.30%, respectively, compared to ARM_dual. For all the applications, except *2dconv*, *histogram*, and *mac*, *DSA_energy* consumed the least energy compared to *M_dsa* and LOCUS, while LOCUS had lowest energy consumption for those applications. LOCUS outperformed the other architectures for these workloads because the workloads were highly parallelizable. As such, LOCUS's 16-core architecture achieved much higher latency reduction, resulting in higher energy savings. We also observed that *DSA_energy* consumed lower energy than *M_dsa* for all the applications, even though *M_dsa* had better performance than *DSA_energy*. This was because the performance improvement from *M_dsa* was not sufficient to offset the increase in power consumption imposed by the dual-core system compared to the single-core *DSA_energy*.

7.4 Area Analysis and Overheads

In this section, we present the area for all the DSA designs in comparison to prior work. To enable a fair evaluation, we estimated the area of LOCUS at 22nm technology node to enable consistency across all the architectures. Using the 22nm node, we estimated LOCUS's area to be 16.47 mm^2 as opposed to the area of 36 mm^2 at 32nm node, which was used in the original work [3]. Figure 14 presents our area analysis, with a breakdown of the different components of the different architectures. We have omitted the LOCUS area from the figure to maintain the scale of the figure. The area of ARM_dual, *DSA_vect*, *DSA_buff*, *DSA_prefetch* and LOCUS were 2.15x, 1.31x, 1.32x, 1.35x and 14.98x larger than the base ARM processor. Importantly, one of the key benefits of our work compared to prior work is the substantial area reduction, without a concomitant energy or latency increase.

We also found that using STT-RAM memories in our DSA design schemes was an important source of additional area benefits. Replacing the SRAM memories with STT-RAMs reduced the memory area by 47.83% and reduced the overall processor area by 20.18% compared to the SRAM-based processor. The area of the 16-bit SIMD execution unit was 1.18x that of the 32-bit scalar execution unit. The area overheads of the SIMD execution unit, buffer, and prefetcher units were 0.58 mm^2, 0.06mm^2 and 0.0266mm^2, respectively. The power requirement of the SIMD execution unit was 1.75x that of the scalar execution unit. The average power overhead of the SIMD execution unit, buffer and prefetcher were 0.0285 W, 0.00714 W, and 0.00744 W, respectively.

7.5 Selecting the Best Design for a Set of Workloads

We have proposed various domain-specific architectures to optimize a range of applications and explored the tradeoffs between the different architectures. A question that may arise is whether or not there is an ultimate design choice given a set of workloads. Clearly, the answer to this question depends on the tradeoffs or design constraints of the target system, and on what kind of applications will run on the wearable devices. For example, if a designer aims to achieve

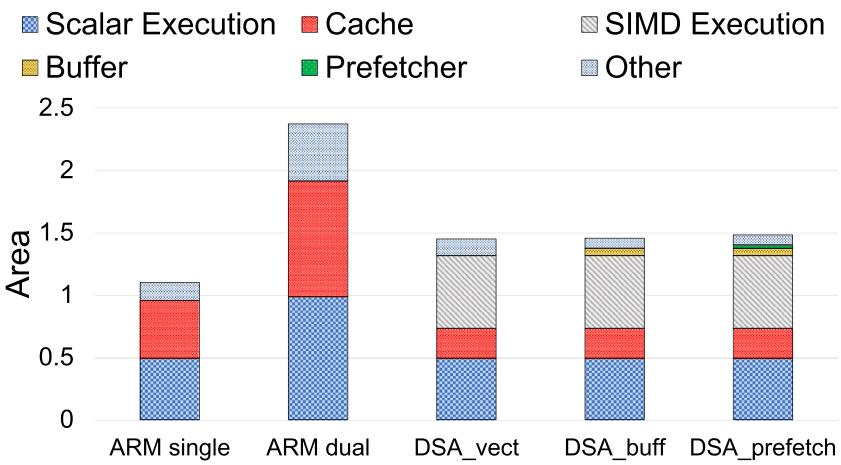

Figure 14 Area of various architectures in mm^2.

the least area overhead, potentially at the expense of energy or latency optimization, *DSA_vect* would be the best choice. If alternatively, the wearable device is dominated by data-rich workloads, like ECG authentication, the performance gained by incorporating the additional buffer featured in *DSA_buff* may be worth the additional area overhead imposed by the buffer. Similarly, if the workloads are dominated by graph applications, then the designers can opt for *DSA_prefetch*. If the ultimate goal is to minimize energy consumption, *DSA_energy* would be the best choice.

8 Conclusion

Domain-specific architectures afford execution flexibility that is unavailable in ASICs and also enable high optimization potential with minimal resource wastage compared to general-purpose architectures. In this paper, we explored domain-specific architectures for resource-constrained wearable computing. To identify the appropriate domain-specific optimizations, we analyzed various wearable workloads to identify their bottlenecks, intrinsic parallelism, data movement, and memory-boundedness. Based on the analysis, we performed design space exploration of a low-power SIMD unit to exploit the available data-level parallelism, an STT-RAM buffer to reduce data movement and computations, and a prefetch unit to preemptively fetch non-contiguous data elements into contiguous memory locations to enhance their parallelism. We analyzed how these optimizations could be incorporated into various domain-specific designs and proposed four DSA design schemes to satisfy different resource requirements. For each of our designs, we analyzed their performance, energy, and area tradeoffs. We also detailed the benefits and tradeoffs of the domain-specific architectures in comparison to a general-purpose base processor and prior art in wearable computing architecture optimizations. Our experimental results show that there is much optimization benefit, in the context of resource-constrained wearable devices, to specializing the computing resources to the wearable workload requirements via domain-specific architectures.

References

1. Park, S., Chung, K., & Jayaraman, S. (2014). Wearables: fundamentals, advancements, and a roadmap for the future. In *Wearable sensors* (pp. 1–23). Elsevier.
2. eservices report 2020 - fitness. [Online]. Available: https://www.statista.com/study/36674/fitness-report/.
3. Tan, C., Kulkarni, A., Venkataramani, V., Karunaratne, M., Mitra, T., & Peh, L.-S. (2017). Locus: Low-power customizable many-core architecture for wearables. *ACM Transactions on Embedded Computing Systems (TECS)*, *17*(1), 1–26.
4. Liu, R., & Lin, F. X. (2016). Understanding the characteristics of android wear os. In *Proceedings of the 14th annual international conference on mobile systems, applications, and services* (pp. 151–164).
5. Hennessy, J. L., & Patterson, D. A. (2019). Computer architecture: a quantitative approach.
6. Cordeiro, R., Gajaria, D., Limaye, A., Adegbija, T., Karimian, N., & Tehranipoor, F. (2020). Ecg-based authentication using timing-aware domain-specific architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *39*(11), 3373–3384.
7. Jouppi, N. P., Young, C., Patil, N., & Patterson, D. (2018). A domain-specific architecture for deep neural networks. *Communications of the ACM*, *61*(9), 50–59.
8. Jouppi, N. P., Yoon, D. H., Kurian, G., Li, S., Patil, N., Laudon, J., Young, C., & Patterson, D. (2020). A domain-specific supercomputer for training deep neural networks. *Communications of the ACM*, *63*(7), 67–78.
9. Kuan, K., & Adegbija, T. (2019). Energy-efficient runtime adaptable l1 stt-ram cache design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *39*(6), 1328–1339.
10. Gotlibovych, I., Crawford, S., Goyal, D., Liu, J., Kerem, Y., Benaron, D., Yilmaz, D., Marcus, G., & Li, Y. (2018). End-to-end deep learning from raw sensor data: Atrial fibrillation detection using wearables, arXiv:1807.10707.
11. Janarthanan, R., Doss, S., & Baskar, S. (2020). Optimized unsupervised deep learning assisted reconstructed coder in the on-nodule wearable sensor for human activity recognition. *Measurement*, *164*, 108050.
12. Wiechert, G., Triff, M., Liu, Z., Yin, Z., Zhao, S., Zhong, Z., Zhaou, R., & Lingras, P. (2016). Identifying users and activities with cognitive signal processing from a wearable headband. In *2016 IEEE 15th International conference on cognitive informatics & cognitive computing (ICCI* CC)* (pp. 129–136). IEEE.
13. Ren, Y., Xie, X., Li, G., & Wang, Z. (2016). Hand gesture recognition with multiscale weighted histogram of contour direction normalization for wearable applications. *IEEE Transactions on Circuits and Systems for Video Technology*, *28*(2), 364–377.
14. Liu, Y., Jiang, F., & Gowda, M. (2020). Application informed motion signal processing for finger motion tracking using wearable sensors. In *ICASSP 2020-2020 IEEE International conference on acoustics, speech and signal processing (ICASSP)* (pp. 8334–8338). IEEE.
15. Kale, N., Lee, J., Lotfian, R., & Jafari, R. (2012). Impact of sensor misplacement on dynamic time warping based human activity recognition using wearable computers. In *Proceedings of the conference on wireless health* (pp. 1–8).
16. Rong, L., Jianzhong, Z., Ming, L., & Xiangfeng, H. (2007). A wearable acceleration sensor system for gait recognition, in 2007 2nd. In *IEEE conference on industrial electronics and applications* (pp. 2654–2659). IEEE.
17. Sundararajan, D. (2011). Fundamentals of the discrete haar wavelet transform.
18. Majmudar, C. A., & Morshed, B. I. (2016). Autonomous oa removal in real-time from single channel eeg data on a wearable device using a hybrid algebraic-wavelet algorithm. *ACM Transactions on Embedded Computing Systems (TECS)*, *16*(1), 1–16.
19. Park, C., Chou, P. H., Bai, Y., Matthews, R., & Hibbs, A. (2006). An ultra-wearable, wireless, low power ecg monitoring system. In *2006 IEEE biomedical circuits and systems conference* (pp. 241–244). IEEE.
20. Braojos, R., Mamaghanian, H., Dias, A., Ansaloni, G., Atienza, D., Rincón, F. J., & Murali, S. (2014). Ultra-low power

design of wearable cardiac monitoring systems. In *2014 51st ACM/EDAC/IEEE design automation conference (DAC)* (pp. 1–6). IEEE.
21. Dieffenderfer, J., Goodell, H., Mills, S., McKnight, M., Yao, S., Lin, F., Beppler, E., Bent, B., Lee, B., Misra, V., & eta l (2016). Low-power wearable systems for continuous monitoring of environment and health for chronic respiratory disease. *IEEE Journal of Biomedical and Health Informatics*, *20*(5), 1251–1264.
22. Dogan, A. Y., Constantin, J., Ruggiero, M., Burg, A., & Atienza, D. (2012). Multi-core architecture design for ultra-low-power wearable health monitoring systems. In *2012 Design, automation & test in europe conference & exhibition (DATE),* (pp 988–993). IEEE.
23. Ickes, N., Sinangil, Y., Pappalardo, F., Guidetti, E., & Chandrakasan, A.P. (2011). A 10 pj/cycle ultra-low-voltage 32-bit microprocessor system-on-chip. In *2011 Proceedings of the ESSCIRC (ESSCIRC)* (pp. 159–162). IEEE.
24. Jouppi, N. P., Young, C., Patil, N., & Patterson, D. (2018). A domain-specific architecture for deep neural networks. *Communications of the ACM*, *61*(9), 50–59.
25. Cong, J., Guruaj, K., Huang, M., Li, S., Xiao, B., & Zou, Y. (2011). Domain-specific processor with 3d integration for medical image processing. In *ASAP 2011-22nd IEEE International conference on application-specific systems, architectures and processors* (pp. 247–250). IEEE.
26. Di Tucci, L., Baghdadi, R., Amarasinghe, S., & Santambrogio, M.D. (2020). Salsa: a domain specific architecture for sequence alignment. In *2020 IEEE International Parallel and distributed processing symposium workshops (IPDPSW)* (pp. 147–150). IEEE.
27. Xin, G., Han, J., Yin, T., Zhou, Y., Yang, J., Cheng, X., & Zeng, X. (2020). Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture. In *IEEE transactions on circuits and systems I: regular papers*.
28. Jain, A. K., Omidian, H., Fraisse, H., Benipal, M., Liu, L., & Gaitonde, D. (2020). A domain-specific architecture for accelerating sparse matrix vector multiplication on fpgas. In *2020 30th International conference on field-programmable logic and applications (FPL)* (pp. 127–132). IEEE.
29. Muzaffar, S., & Elfadel, I. M. (2019). A domain-specific processor microarchitecture for energy-efficient, dynamic iot communication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *27*(9), 2074–2087.
30. Waheed, O. T., & Elfadel, I. A. M. (2019). Domain-specific architecture for imu array data fusion. In *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)* (pp. 129–134). IEEE.
31. Reinders, J. (2005). Vtune performance analyzer essentials. Intel Press.
32. Thiel, J. (2006). An overview of software performance analysis tools and techniques: From gprof to dtrace, Washington University in St. Louis, Tech. Rep.
33. Tanaka, H., Ota, Y., Matsumoto, N., Hieda, T., Takeuchi, Y., & Imai, M. (2010). A new compilation technique for simd code generation across basic block boundaries. In *2010 15th Asia and South pacific design automation conference (ASP-DAC)* (pp. 101–106). IEEE.
34. Karrenberg, R. (2015). Whole-function vectorization. In *Automatic SIMD vectorization of SSA-based control flow graphs* (pp. 85–125). Springer.
35. Shahbahrami, A., Juurlink, B., & Vassiliadis, S. (2007). Simd vectorization of histogram functions. In *2007 IEEE International conf. on application-specific systems, architectures and processors (ASAP)* (pp. 174–179). IEEE.
36. Chang, H., & Sung, W. (2008). Efficient vectorization of simd programs with non-aligned and irregular data access hardware. In *Proceedings of the 2008 international conference on compilers, architectures and synthesis for embedded systems,* (pp. 167–176).
37. Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., & et al. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*, *39*(2), 1–7.
38. Raman, S. K., Pentkovski, V., & Keshava, J. (2000). Implementing streaming simd extensions on the pentium iii processor. *IEEE Micro*, *20*(4), 47–57.
39. Pennycook, S. J., Hughes, C. J., Smelyanskiy, M., & Jarvis, S.A. (2013). Exploring simd for molecular dynamics, using intel® xeon® processors and intel® xeon phi coprocessors. In *2013 IEEE 27th International symposium on parallel and distributed processing* (pp. 1085–1097). IEEE.
40. Spracklen, L., & Abraham, S. G. (2005). Chip multithreading: Opportunities and challenges. In *11th International symposium on high-performance computer architecture* (pp. 248–252). IEEE.
41. Olszewski, M., Ansel, J., & Amarasinghe, S. (2009). Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th international conference on architectural support for programming languages and operating systems* (pp 97–108).
42. Sun, Z., Bi, X., Li, H., Wong, W.-F., Ong, Z.-L., Zhu, X., & Wu, W. (2011). Multi retention level stt-ram cache designs with a dynamic refresh scheme.
43. Smullen, C. W., Mohan, V., Nigam, A., Gurumurthi, S., & Stan, M.R. (2011). Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *2011 IEEE 17th International symposium on high performance computer architecture* (pp 50–61). IEEE.
44. Qiu, H., Wang, X., & Xie, F. (2017). A survey on smart wearables in the application of fitness. In *2017 IEEE 15th Intl conf on dependable, autonomic and secure computing, 15th intl conf on pervasive intelligence and computing, 3rd intl conf on big data intelligence and computing and cyber science and technology congress (DASC/PiCom/DataCom/CyberSciTech)* (pp. 303–307). IEEE.
45. Duncan, R. (1990). A survey of parallel computer architectures. *Computer*, *23*(2), 5–16.
46. Firasta, N., Buxton, M., Jinbo, P., Nasri, K., & Kuo, S. (2008). Intel avx: New frontiers in performance improvements and energy efficiency. *Intel White Paper*, *19*, 20.
47. Reddy, V. G. (2008). Neon technology introduction. *ARM Corporation*, *4*, 1.
48. Fatemi, H., Corporaal, H., Basten, T., Kleihorst, R., & Jonker, P. (2005). Designing area and performance constrained simd/vliw image processing architectures. In *International conference on advanced concepts for intelligent vision systems* (pp. 689–696). Springer.
49. Fijany, A., & Hosseini, F. (2011). Image processing applications on a low power highly parallel simd architecture. In *2011 Aerospace conference* (pp. 1–12). IEEE.
50. Fabietti, P., Benedetti, M. M., Bronzo, F., Reboldi, G., Sarti, E., & Brunetti, P. (1991). Wearable system for acquisition, processing and storage of the signal from amperometric glucose sensors. *The International Journal of Artificial Organs*, *14*(3), 175–178.
51. Yamaguchi, T., Mikami, S., Saito, M., Okada, K., & Gotouda, A. (2018). A newly developed ultraminiature wearable electromyogram system useful for analyses of masseteric activity during the whole day. *Journal of Prosthodontic Research*, *62*(1), 110–115.
52. Park, E., Kim, D., & Yoo, S. (2018). Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *2018 ACM/IEEE 45th Annual international symposium on computer architecture (ISCA)* (pp 688–698). IEEE.

53. Lee, S. Y., & Lee, K. (2018). Factors that influence an individual's intention to adopt a wearable healthcare device: The case of a wearable fitness tracker. *Technological Forecasting and Social Change*, *129*, 154–163.
54. Oliver, N., & Flores-Mangas, F. (2006). Healthgear: a real-time wearable system for monitoring and analyzing physiological signals. In *International workshop on wearable and implantable body sensor networks (BSN'06)* (pp. 4–pp). IEEE.
55. Nakhkash, M. R., Gia, T. N., Azimi, I., Anzanpour, A., Rahmani, A. M., & Liljeberg, P. (2019). Analysis of performance and energy consumption of wearable devices and mobile gateways in iot applications. In *Proceedings of the international conference on omni-layer intelligent systems,* (pp. 68–73).
56. Coke, J. S., Bhatt, A. V., Graham, S., & Lent, D. (1998). Implementing scatter/gather operations in a direct memory access device on a personal computer, Jan. 13 1998, uS Patent 5,708,849.
57. Strey, A., & Bange, M. (2001). Performance analysis of intel's mmx and sse: A case study. In *European conference on parallel processing*(pp. 142–147). Springer.
58. Limaye, A., Tumeo, A., & Adegbija, T. (2020). Energy characterization of graph workloads. Sustainable Computing: Informatics and Systems 100465.
59. Cherupalli, H., Duwe, H., Ye, W., Kumar, R., & Sartori, J. (2017). Enabling effective module-oblivious power gating for embedded processors. In *2017 IEEE International symposium on high performance computer architecture (HPCA)* (pp. 157–168). IEEE.
60. A. Ltd., Arm development studio: Streamline performance analyzer. [Online]. Available: https://developer.arm.com/tools-and-software/embedded/arm-development-studio/components/streamline-performance-analyzer.
61. Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., & et al. (2017). The arm scalable vector extension. *IEEE Micro*, *37*(2), 26–39.
62. Waterman, A. S. (2016). Design of the risc-v instruction set architecture, Ph.D. dissertation, UC Berkeley.
63. Dong, X., Xu, C., Xie, Y., & Jouppi, N.P. (2012). Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *31*(7), 994–1007.
64. Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., & Jouppi, N.P. (2009). Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM international symposium on microarchitecture,* pp. 469–480.
65. Feist, T. (2012). Vivado design suite. *White Paper*, *5*, 30.
66. Dagum, L., & Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, *5*(1), 46–55.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.