WearCore: A Core for Wearable Workloads

Sanyam Mehta*
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL USA
sanyam.mehta@gmail.com

Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL USA
torrella@illinois.edu

ABSTRACT

Lately, the industry has recognized immense potential in wearables (particularly, smartwatches) being an attractive alternative/supplement to the smartphone. To this end, there has been recent activity in making the smartwatch 'self-sufficient' i.e. using it to make/receive calls, etc. independently of the phone. This marked shift in the way wearables will be used in future calls for changes in the core micro-architecture of smartwatch processors.

In this work, we first identify ten key target applications for the smartwatch users that the processor must be able to quickly and efficiently execute. We show that seven of these workloads are inherently parallel, and are compute- and data-intensive. We therefore propose to use a multi-core processor with simple out-of-order cores (for compute performance) and augment them with a light-weight software-assisted hardware prefetcher (for memory performance). This simple core with the light-weight prefetcher, called WearCore, is 2.9x more energy-efficient and 2.8x more area-efficient over an in-order core. The improvements are similar with respect to an out-of-order core.

1. INTRODUCTION

Smartwatches, with the potential to perform many of the important functions of smartphones while just being a band on the wrist, are being deemed as the future of consumer electronics. A recent prediction from ABI Research claims that there will be 485 million annual wearable shipments by 2018 [14]; the prediction is backed up by pronounced activity in the industry with Samsung releasing its sixth smartwatch within a span of one year.

Although an attractive supplement/substitute to smartphones on paper, smartwatches have not yet proved to the purpose. The primary reason for this disparity is that smartwatches currently in the market are not 'self-sufficient', i.e. they require to be tethered to a phone (from perhaps the same manufacturer) in close vicinity (via Bluetooth) in order to make/receive calls and be connected to the internet. This greatly limits the utility of a smartwatch since one has to have the phone around always anyway. However, Google and Samsung have overcome this drawback in existing smartwatches, through their latest offerings. Google's latest update to their version of Android Operating System, Android Wear, enables Wi-fi connectivity in the watch that allows it to connect to a distant phone [25]. Samsung, on the other hand, has gone a step further in its latest Gear S smartwatch (that runs Samsung's own Tizen OS) which can house its own SIM card and has built-in 3G connectivity [12], i.e. it can independently make/receive calls, and send/receive messages or mails. Thus, the current trend in industry is towards 'self-sufficient' smartwatches that obviates the need of carrying or being in the vicinity of a smartphone.

While there have been advances in the OS (and 'apps') for the smartwatch, there has been little contribution from computer architects towards the right hardware for the purpose. We identify the following challenges in coming up with that right hardware.

- Firstly, it is important to identify the applications that are critical for smartwatch users. With a significantly different power (and energy) budget, display size and usage scenario from existing smartphones, it is intuitive to expect that a different set of applications will become important to smartwatch users than smartphone users. For example, automatic speech-to-text (and also speech-to-command, as in an Intelligent Personal Assistant like Siri or Google Now) conversion is extremely important in a smartwatch given the extreme difficulty in typing on the watch. Also, this speech-to-text conversion must happen on the device to keep the device functional when not connected to internet. This 'self-sufficiency' is much more important for a smartwatch than a smartphone since the former cannot continue to use the power-expensive 3G service for longer duration.

- With the target applications known, the host processor must be able to efficiently execute them. While energy efficiency has remained critical even in smartphones, it is much more so for the watch for it is constrained to use a much smaller battery and consequently, has a much lower battery life (Samsung's latest watch, Gear S, and phone, Galaxy S6, have battery capacities of 300mAh[1] [10] and 2550mAh [9], respectively). It is therefore imperative to make the right decisions on the type (In-Order(InO), Out-of-Order(OoO), etc.) and number of cores, and also on the specific additional capabilities in the cores so as to meet the Quality of Service (QoS) requirements of critical applications while being as energy-efficient as possible. Currently, the latest smartwatches from the biggest companies display a lot of variety in the type of cores - Samsung, LG and Apple employ a dual-core OoO,

*This author is now with Cray Inc.

PACT '16, September 11-15, 2016, Haifa, Israel
© 2016 ACM. ISBN 978-1-4503-4121-9/16/09...$15.00
DOI: http://dx.doi.org/10.1145/2967938.2967956

[1]The battery life of smartwatches is about an eighth of smartphones. Now, given that a smartwatch will be used less but should last more, we speculate that a power envelope that is a fourth or a sixth of a smartphone seems reasonable for a smartwatch.

a quad-core InO and a single-core OoO processor, respectively. Clearly, the best core for smartwatches is undecided.

In this paper, we make the following contributions,

- We identify ten important applications for smartwatches that the host hardware must accommodate to satisfy the energy-efficiency demands of the consumers. We refer to this set of applications as WearBench throughout the paper. We identify an application as important when it satisfies one or more of the following three criteria, (1) demands a strict QoS, (2) is compute-intensive (and time consuming), and (3) is frequently used. We classify these applications in five categories, (a) speech recognition (speech-to-text and speech-to-command), (b) image processing (image denoising, compression), (c) computer vision (face recognition, image classification), (d) audio playback (mp3), and (e) video rendering (h264 decode). We do not include sensor processing applications in this set since the signals from sensors are efficiently processed in the DSP on the SoC (see Section 3 for more discussion).

- For applications in WearBench, we make the following important observations. (1) 7 out of 10 applications (including applications with a QoS requirement) are parallel and benefit immensely from multiple cores on the chip. (2) These applications are compute and data intensive and not so control intensive. Moreover, 25% of all operations in these applications are SIMD operations on average (and the applications in categories (a), (b) and (c) have fully vectorizable critical loops). As a result, computation in loops finishes very quickly, putting more pressure on the memory. Thus, cache misses become significant. This trend will only become more significant as the industry considers larger vector lengths to speed up applications as in WearBench.

- We make the following decisions based on above observations. (1) We use a quad core CPU as our baseline processor to allow parallel execution. (2) Each of the cores supports only partial out-of-order execution (using a simple scoreboard) that only adds little overhead over in-order cores. (3) In order to prevent stalls on long-latency cache misses, we augment our simple OoO core with a software-assisted hardware prefetcher (SAHP). We call this core as WearCore. Our proposed SAHP is easy to use with the target applications, involves negligible instruction overhead and minimal power/area overhead in the hardware.

- We compare the performance, power and area of WearCore with an OoO and an InO core for applications in WearBench. WearBench proves to be about 2.8x better in terms of area-normalized performance and about 2.9x better in terms of energy-efficiency with respect to both InO and OoO cores.

It is important to note that we do not claim novelty in either WearBench (since we obtain the component applications from other suites) or the choice of core (since scoreboarding for partial out-of-order execution is already known) or the number of cores. Our key contribution is in identifying the advantages and bottlenecks of these choices for the new problem domain of wearables (which we believe has not been explored before), and in our proposal of SAHP to address the last standing limitation in the design of an energy-efficient core for wearables.

The rest of the paper is organized as follows. Section 2 describes each of the ten applications used in WearBench with particular emphasis on the critical speech recognition applications. In Section 3, we discuss why other possible design options were incompatible with WearBench and present our key insights that motivate our specific choices in the design of WearCore. This section also provides an overview of SAHP. The following section details the implementation of SAHP in WearCore. Section 5 describes the microarchitectural details of the different cores used for the purpose of comparison, and also how we use the applications with the microarchitectural simulator used. We present and discuss the results in Section 6; this includes performance comparison, and area and power estimates of the different configurations of cores simulated. We also show the parallel performance achieved by two representative applications in WearBench, and how exploiting parallelism helps them to meet their QoS. Section 7 briefly describes other work related to ours, and we present the conclusion from this work in Section 8.

2. TARGET APPLICATIONS

As mentioned in Section 1, WearBench consists of applications in seven different categories. In this section, we briefly describe those applications and their relevance to a smartwatch.

2.1 Automatic Speech Recognition (ASR)

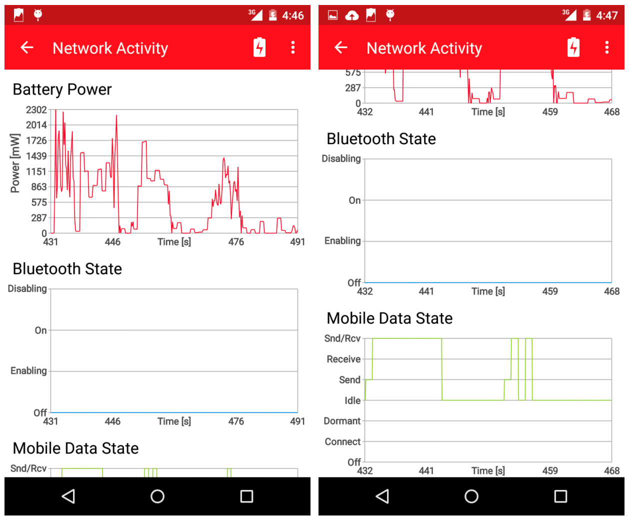

Figure 1: Power profile (left) and mobile data state (right) of Google Nexus3 phone using the Qualcomm Trepn profiler when executing a command through Google Now

This category represents a set of key applications for smartwatch users. The reason for this is that a smartwatch is much more typing-unfriendly than a smartphone due to a tiny screen, and the user has to spend a lot of time to type messages and additionally, rely heavily on automatic correction. This has proved to be very unsatisfactory [13] and therefore future smartwatches will have to rely heavily on automatic speech recognition to (1) convert speech to text to facilitate sending of messages, etc., and (2) convert speech to actions or commands that need to be executed on the watch using an Intelligent Personal Assistant (IPA) like Siri, Google Now, etc. Recently, there has been a lot of interest in offline voice recognition on a smartphone. Google recently developed an accurate, small-footprint, large vocabulary speech recognizer for smartphones that uses state-of-the-art deep neural networks (DNNs) as acoustic models [31]. This enables, typing text or issuing commands using speech even when not connected to the Internet, and also fast response in slower networks. While this is important in the context of a smartwatch too for its self-sufficiency, offline speech recognition is more important from a power/energy perspective on a smartwatch.

Figure 1 shows the power profile and the corresponding change of mobile data state on a phone that is using 3G connection to execute a voice command through the Google's IPA, Google Now. The figure shows that even for a query that is executed and responded to in a few seconds (usually less than 2 seconds), the mobile device has to be in a high power Snd/Rcv state for around 12 seconds, during which it consumes 1200mW of average power and 2300mW peak power as shown in the figure. A DNN-based offline speech recognizer can execute the same query in less than a second at 1000mW peak power (with all 4 cores active) [3], thus being 14.4x more energy efficient.

We choose four applications in this category. All of these applications are also part of the Tonic Suite [27] where the authors use them as representative future server workloads. We adapt those applications for a smartwatch. The first application, *speech-to-text*, uses a speech-to-text decoder adapted from Kaldi [40]. In WearBench, we only consider the DNN scoring that houses the DNN computation and is the most time-consuming portion [31]. This is usually followed by some post-processing. The model used is a standard feed forward neural network with k hidden layers of n_h nodes. Each layer computes a non-linear function of the weighted sum of the outputs of the previous layer. We use the same parameters in our model as in [31], i.e. there are 640 inputs and 2000 outputs. The input chosen is obtained from a few seconds of speech after some pre-processing to obtain its feature vectors. Further, there are (k=6) hidden layers with (n_h=512) nodes each. The DNN computation entails executing the $sgemm$ subroutine from the BLAS library, i.e. matrix multiplication of single-precision floats. Using ARM Streamline performance analyzer [4] on Cortex-A15, we find that multiple $sgemm$ instances together account for 95% of the execution time in DNN scoring. Table 1 shows the size of the largest matrices that are multiplied in different layers of the neural network.

The other three applications are Natural Language Processing (NLP) tasks that gather semantic information from input text. The three applications are, (1) part-of-speech (pos) tagging that assigns a tag to each word of speech (i.e. noun or verb), (2) word chunking (chk) that tags different segments of a sentence as a noun or verb phrase, and (3) name entity recognition (ner) that assigns a label to each word as to whether it is a person or a location; all these three applications are based on Senna [20]. These applications combine with speech-to-text to provide the speech-to-command functionality as needed in an IPA. Using Streamline, we find that the $sgemm$ subroutine within the DNN computation accounts for 68%, 70% and 74% of the execution time, respectively, in the *pos*, *chk* and *ner* applications. Again, the size of the largest matrices that are multiplied in different layers of the neural network is given in Table 1. The input chosen is a vector of 10 words of speech (as can be uttered in around 5 seconds to an IPA) as opposed to a 28-word vector in the Tonic Suite.

Since *sgemm* forms the core of these four applications, we use a very fast implementation of *sgemm*. We make it fast by (1) parallelizing the outermost loop to utilize the multiple cores on the chip, (2) tiling or blocking the matrices as suggested in recent papers [34, 36] that reuses data in both L1 and L2 cache and is also more amenable to prefetching, (3) unrolling the loops and finding the best unroll factors on real hardware, and (4) using ARM neon-vfpv4 SIMD extensions. We use different tile sizes and unroll factors for different sizes of the matrices. For the *sgemm* kernel used in *ASR*, our implementation performs 35% better than that in the OpenBLAS library [8] for ARM Cortex-A15 when tested on real hardware, both for a single-thread and multiple threads.

Category	Application	Layers	Size of matrices
ASR	speech-to-text	6	548x512x512, 548x2000x512
	pos	3	10x300x300
	chk	3	10x300x300
	ner	3	10x300x375
Computer Vision	imc	22	3x4096x4096, 3x4096x9216
	face	7	32x20164x363, 16x3969x2592

Table 1: Configuration of the neural networks used in WearBench applications

2.2 Image Processing

WearBench includes two applications in this category. One of them is the standard JPEG compression (*jpegc*), and the other is image denoising (*denoising*). The latter is particularly relevant to a smartwatch where the camera lens being thinner than that in a smartphone, cannot take sharp images. Thus, image processing applications such as image denoising that attenuate the noise added during image acquisition, can greatly improve the final image quality. *jpegc* involves first reading blocks of 8x8 pixels from the image, and the data in each block undergoes a sequence of operations including discrete cosine transform (DCT) and others. For image denoising, we use a simple and effective algorithm [46] that also involves reading the data in the image into overlapping (or sliding) 16x16 patches of pixels, and then applying DCT followed by inverse-DCT to obtain the denoised image. As in *sgemm*, we unroll the loops in DCT by the best unroll factors found on Cortex-A15, and also vectorize the inner loop.

2.3 Computer Vision

We include two applications in this category that are representative of some forward-looking uses of a smartwatch. The first is the face recognition (*face*). This application is particularly relevant for users as it replaces the task of entering password on the watch (as needed for some of the very important websites as mail, etc.) with a mere glance at the watch asking it to recognize the user's face (see [11] for an example application). This is useful since the user cannot also speak his/her password to sign in, when amidst people. The second application in this category is the image classification (*imc*). This application helps users to organise images taken as reminders, letting them add notes, tags and location data by predicting what the image contains. Both the face recognition and image classification applications are used from the Tonic Suite which contains implementations of these based on Facebook's DeepFace [43] facial recognition network and AlexNet [30], a large scale image classifier. The number of layers and the size of the matrices in the DNN computation within the hidden layers in these two applications is given in Table 1.

2.4 Audio Playback

This is an important category of applications for any smart device. This is particularly relevant for a smartwatch as users are expected to rely heavily on this for music, and it proves to be handy for hearing music when out for a workout, etc. We include a simple mp3 player, mpeg3play [7], in this category. We chose this player particularly because it includes optimizations to improve speed and we find that 14% of all operations executed in the player are vector operations. This application spends most of the execution time in decoding audio frames of an input mp3 file. The decoder source code contains a mix of memory access and arithmetic operations, but the memory footprint of the application is low.

2.5 Video Rendering

This category contains CISCO's open-source implementation of the H264 decoder application. The H264 decoder is used to decode frames from an input video stream and render it on the host. This is useful to allow video streaming on a smartwatch, which would require this capability once equipped with 3G service. Like the mp3 player, most of the execution time is spent on decoding input frames from the stream. Unlike the mp3 player, the memory footprint of the application is larger, and thus efficiently fetching data from memory is critical.

3. MOTIVATION AND KEY INSIGHT

This section presents key insights that lead to our specific design choices and motivates those choices over other possible solutions.

3.1 Why not a DSP for WearBench?

Digital Signal Processors (DSPs) effectively complement CPUs in a mobile SOC by serving as low-power accelerators (with VLIW cores) for audio and video processing. Recently, DSPs have been shown to efficiently handle conditional code and also tolerate (some) memory latency through the use of simultaneous multithreading such as in Qualcomm's Hexagon DSPs [18] that employ three threads in the DSP. It is for these reasons that DSPs are proving effective for processing data from sensors such as in various fitness tracking applications in a smartwatch. However, DSPs do not support multiprocessing in hardware (or software), i.e. a DSP uses only one core since there is not much inherent parallelism in traditional DSP workloads. The DSPs do not support fine-grain parallelism either for similar reasons. As a result, it is not suitable for handling applications in WearBench, most of which have inherent parallelism (coarse- as well as fine-grain). This parallelism must be exploited to minimize the response latency of critical user applications such as speech recognition in order to enhance QoS; the improvement of communication interface in smartwatches such as via speech, etc. has been recently recognized as a key factor governing adoption of this new technology [44].

3.2 Why not an Accelerator?

Since most of the important applications essentially employ a neural network and hence involve a multiplication of matrices (albeit of different sizes for different applications), it appears lucrative to employ an accelerator dedicated to matrix multiplication in a smartwatch. However, there are two drawbacks of this strategy.

1. The design of an accelerator takes long design cycles and requires extensive expertise in hardware design. In the meantime, the algorithms for some of the interesting applications for the smartwatch platform are changing. An example is Google's move from Deep to Recurrent Neural Networks [24] which provide better accuracy in speech recognition. Another example is Samsung's latest introduction of user's hand-gesture recognition for interfacing with the smartwatch. Thus, employing specialization in this emerging domain is risky; we therefore argue that an efficient general-purpose core for this platform will be a win in the long run. Also, the strict area and power constraints of a smartwatch SoC disfavor extensive specialization. That is, it is not feasible to have an H264 unit, an audio decoder, a DNN accelerator and a DSP all on a smartwatch SoC; ARM's NEON SIMD unit is meant to prevent over-specialization.

2. A very recent work [33] presents a way to automatically and efficiently generate accelerators (realized using FPGAs) for various machine learning algorithms. However, the solution is not very effective for neural networks and only achieves a performance improvement of 3x with respect to Cortex-A15 (while it achieves an average improvement of 15x over ten machine learning tasks). This is because the different stages of a neural network cannot be executed in parallel, leading to minimal gains with the accelerator. Furthermore, the opportunity for parallelism within each stage is rather limited as can be observed from the small size of the matrices in one of the dimensions (this dimension corresponds to the outermost loop and is split across parallel units) in many applications in Table 1. The paper also points that the neural network task with 512 inputs and outputs and 256 nodes consumes the maximum resources on the FPGA as compared to any other task; we would need even more resources for WearBench.

3.3 Key Insights

Table 2 compares the number of SIMD instructions and the L1 data cache miss rate (**M**isses **p**er **K**ilo **I**nstructions) for WearBench and BBench [26]. The numbers for WearBench are obtained from the gem5 simulator [15]. We rely on [26] for numbers on data cache miss rate since they obtain them on real hardware. From Section 2 and Table 2, we can make the following important observations.

	SIMD ops (%)	L1 Data Cache Miss Rate (MpKI)
BBench	~0	7
WearBench	25	30

Table 2: Comparison between WearBench and BBench

1. Seven of ten applications in WearBench are inherently parallel and can utilize all the cores on the chip. In some applications such as the automatic speech recognition, it is rather imperative to effectively use all the cores on chip in order to meet the QoS requirement.

2. The average miss rate incurred by WearBench is more than 4x higher than that incurred by BBench. WearBench is clearly more data-intensive than BBench. This pronounced data intensity in WearBench stems from the fact that the applications usually continuously load data (rows of matrices in case of speech recognition and computer vision, frames of audio/video in case of audio/video decoding, etc.) from a lower level of memory and then process that data (for example, multiply-and-accumulate in speech recognition, some signal processing algorithms in image/audio/video processing).

3. On average, 25% of the operations utilize the NEON SIMD unit on the core. In contrast, BBench does not utilize the SIMD unit at all; this is usually attributed to complex control flow within loop bodies in browser source code that makes it hard for the compiler to auto-vectorize loops. The applications in WearBench on the other hand contain loops without complex conditionals and therefore benefit from vectorization. As a result of this vectorization, however, the computation in loops executes very fast. This makes cache miss latency critical to overall application performance.

From the above observations, we make the following conclusions.

1. While future smartwatches must employ multiple cores, the cores must be simple to satisfy the low area and power budget of smartwatches. Clearly, employing multiple out-of-order cores will make it very difficult to meet these requirements. Instead, simpler cores that map well to the target applications and yield comparable performance to the out-of-order cores, will provide the maximum energy efficiency.

2. Given the high data cache miss rate in WearBench, an in-order core that stalls on every miss will not meet the QoS requirements of WearBench. Thus, we propose to use as our

% Samples		Disassembly
In-order	Out-of-order	
		.L40
		...
3.33%	**10.34%**	**5:** vldr d18, [r2, #-16] 'Vector load'
7.58%	3.45%	**6:** vldr d19, [r2, #-8]
0.91%	-	**7:** add r1, r1, #16 'Integer add'
	-	...
1.82%	**8.62%**	**10:** vldr d20, [r1, #-16]
5.15%	3.45%	**11:** vldr d21, [r1, #-8]
0.91%	-	**12:** add r0, r0, #16
0.91%	1.72%	**13:** vldr d24, [r5, #-16]
4.24%	**13.79%**	**14:** vldr d25, [r5, #-8]
		...
1.21%	-	**17:** vfma.f32 q7, q10, q9 'Vector fp multiply-add'
5.76%	-	**18:** vfma.f32 q14, q10, q8
		...
6.36%		**27:** bne .L40 'End of loop'

Figure 2: Disassembly of the innermost loop in the *sgemm* kernel and the time spent in individual instructions within in-order and out-of-order cores, as obtained from the ARM Streamline Performance Analyzer on Odroid-XU3 board with in-order ARM Cortex-A7 and out-of-order ARM Cortex-A15 cores

baseline core, a partial out-of-order core that uses a scoreboard to prevent RAW hazards and allows for few outstanding cache misses. However, since there is no renaming and speculation, the core is much simpler than a full out-of-order core. We further address the area and power concerns by using a smaller L2 cache, and by allowing merely two outstanding data cache misses.

3. The use of scoreboarding alone, however, is not sufficient to achieve performance close to the fully out-of-order core. This can be understood from the assembly shown in Figure 2. Consider the loads in lines 10 and 11 each of which loads a double-word. This data is used (as a quad-word) in the vector floating-point multiply-and-accumulate instruction in line 17. A two-wide instruction issue would imply a three cycle latency between load and use. Thus, unless the data is in the L1 cache (2 cycles in case of Cortex-A7 and Cortex-A15), the core using scoreboard will stall to respect this RAW dependence. The impact of an L1 miss can be seen in Figure 2 where both the cores witness considerable latency for all the loads that are serviced from the L2 cache in the chosen *sgemm* kernel. As a result, we find that using the hardware prefetcher to prefetch the data to L1 still gives a 12% performance gain for an out-of-order core. Clearly, this is much more so for our partial out-of-order baseline core. We thus propose to use a light-weight hardware prefetcher in WearCore to prefetch the data to the L1 cache.

3.4 The Case for a Software-Assisted Hardware Prefetcher

The existing ARM cores provide different prefetching options. The ARMv7-A ISA supports software prefetching through the preload instruction, 'PLD [rn, #offset]' where *rn* is the register carrying the base address and *offset* is the optional offset. Also, the out-of-order core variants employ a powerful hardware streamer prefetcher at the L2 cache that supports large prefetch distances (8 cache lines) and can detect multiple data streams. The in-order variants either do not implement hardware prefetching (as in Cortex-A8) or use a simplistic prefetcher at L1 cache that can track just two data streams (as in Cortex-A7 and A9). However, each of these options proves inadequate for applications in WearBench for the following reasons.

1. Using the preload instruction (PLD) in the innermost loop to prefetch contiguous data accesses worsens the performance by 20% due to the overhead of the PLD instruction itself on the in-order Cortex-A7 core. Thus, software prefetching is unsuitable due to the performance overhead.

2. The simplistic hardware prefetcher (that can track two data streams) at the L1 cache proves insufficient also. This is because the compiler unrolls the loops in many of these applications, and thus exposes multiple data streams (8 in *jpegc*, and 6 in *sgemm* and *denoising*) in the loop body. Also, these cores do not employ a sophisticated hardware prefetcher at L1 since the overhead of the hardware structures needed to track multiple streams in terms of storage, area and power, is large. For example, a 256-entry Global History Buffer (GHB) requires 6KB of storage space, and also consequently area and power overheads since GHB entries need to be updated on every data access.

3. Prefetching the data to the L2 cache using a sophisticated hardware prefetcher is not sufficient either to hide latency when using a partial out-of-order core as above discussed.

Thus, we propose to use a software-assisted hardware prefetcher (SAHP) that combines the advantages of both the software and hardware prefetching and avoids their respective drawbacks. SAHP is triggered by our proposed software prefetch instruction, PLDX, as **PLDX [rn, #nl]**, where *rn* carries the base address as in PLD, and *nl* is the number of cache lines that should be prefetched starting at that address. This has the following advantages.

1. Since it is outside the loop, its contribution to the instruction overhead is negligible.

2. Since the information about what data to prefetch is entirely encapsulated in this instruction, there is no sophisticated hardware needed to track data streams.

3. Once the prefetching is triggered, the hardware assumes complete responsibility to maintain the appropriate prefetch distance such that it hides the latency effectively and also does not cause cache pollution. The latter is important since SAHP prefetches the data all the way to the (smaller) L1 cache for effective latency hiding. Thus, SAHP relieves the user from setting the right prefetch distance, which is known to be difficult to determine.

4. Finally, since the hardware knows well in advance about the number of cache lines that need to be prefetched (possibly across virtual page boundaries), it leverages this information to prefetch across physical pages and thus does not have to stop at physical page boundaries.

It is important to note that the user or the library developer is responsible for the insertion of these prefetch instructions into the code to trigger SAHP. However, the insertion of these instructions is extremely simple for the programmer. In many of the WearBench applications, just a prefetch instruction before a subroutine call or loop body is sufficient since this new prefetch instruction can capture all data needed in the loop body (in the form of the number of the lines to be prefetched that is passed as the second argument in the 'PLDX' instruction) at once. For example, in *denoising*, the three suborutine calls each needed just a single prefetch instruction for the entire subroutine because it contained the one array that was responsible for memory accesses; the number of lines to be prefetched is specified as a function of the image size and is therefore not necessary to be determined at compile time.
We detail the implementation of SAHP in the following section.

Structure	Size	Location	Functionality
prefetch bit	1 bit	Each block in L1 cache	Indicates if this block was prefetched when brought to cache
read bit	1 bit	”	Indicates if this block has been read at least once after being prefetched
mID	4 bits	”	Holds the ID of the MSHR that is servicing the prefetch request responsible for fetching this block to the cache
counter	4 bits	Each MSHR in MSHR file	Contains the count of prefetched blocks that have not yet been read
stop bit	1 bit	”	This bit is set to true when the counter reaches its maximum value of 15, to indicate that this MSHR should stop prefetching further for now
num	16 bits	”	Contains the count of the number of blocks prefetched through this MSHR that have also been read
nblocks	16 bits	”	Contains the number of blocks that remain to be prefetched from the original number of requested blocks in the prefetch request
ID	4 bits	”	Unique ID for this MSHR
Vaddr	20 bits	”	Contains the most significant 20 bits of the virtual address of the block being prefetched

Table 3: Hardware structures added to implement SAHP; SAHP adds an overhead of 6 bits per cache line and 61 bits per MSHR at L1 cache

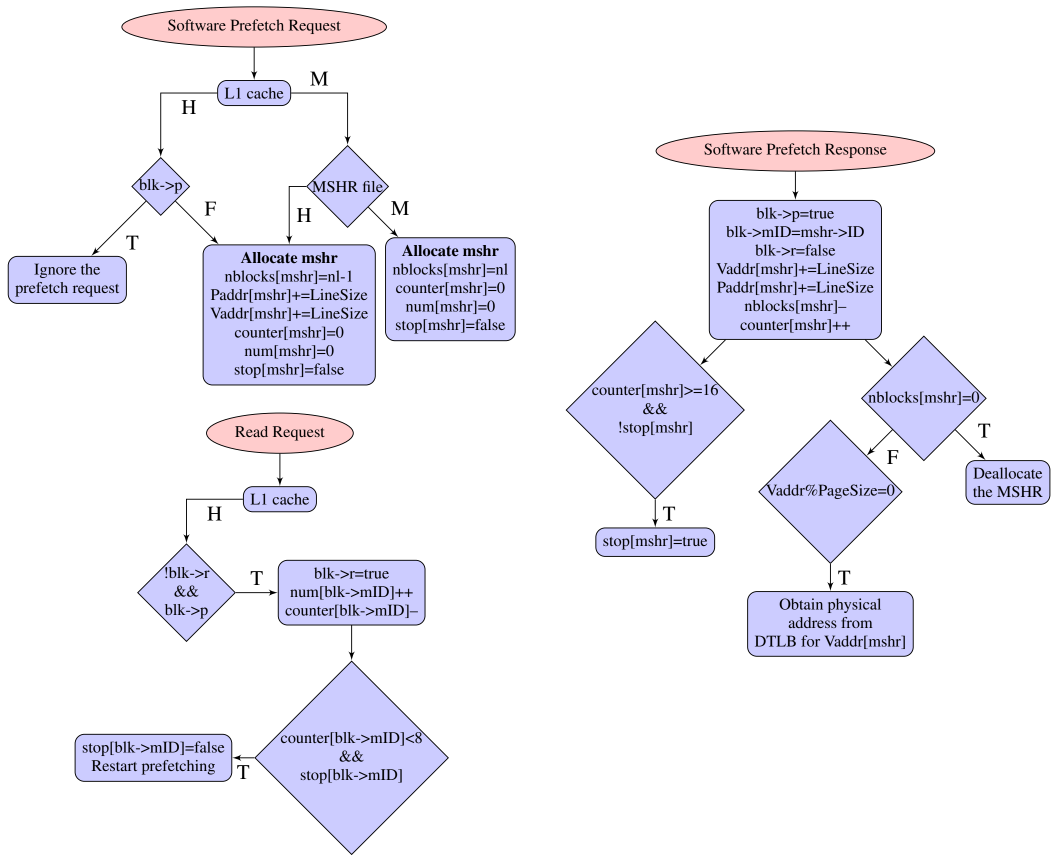

Figure 3: Actions taken at prefetch/read request and response

4. IMPLEMENTATION

Our proposed software-assisted hardware prefetcher (SAHP) is designed to complement our baseline partial out-of-order core for future smartwatches. The core uses scoreboarding for out-of-order execution where we allow a maximum of two outstanding cache misses at any time. That is, if all data is found in the L1 data cache that has a hit latency of 2 cycles, data reads can proceed without stalls. This strategy is similar to Intel's Xeon Phi coprocessor where each thread in the core can issue a load instruction once in two cycles because another load cannot issue unless the data for the first load is returned from the L1 cache (with 2 cycle hit latency). Therefore, like Xeon Phi, our baseline core relies heavily on data having been prefetched to the L1 cache to successfully hide latency.

As mentioned earlier, SAHP is triggered by a software prefetch instruction that specifies the number of cache lines to be prefetched. Once triggered, SAHP prefetches the specified number of lines to the L1 cache through the use of some extra hardware as listed in Table 3. Since SAHP bears the onus to maintain the right prefetch distance, the hardware must provide a mechanism to stop/restart prefetching while the program is executing. This is crucial because in many programs (such as *jpegc* and *denoising* in WearBench), there is intermittent fetch and compute, and the prefetcher could easily run far ahead during the compute phase causing cache pollution if it is not timely stopped.

In SAHP, a single software prefetch instruction instructs the hardware to prefetch all the data needed by a particular data stream. Thus, our 'PLDX' software prefetch instruction remains bound to a single MSHR until all needed cache lines have been prefetched. Each MSHR is therefore equipped with its own counter (and stop bit) to determine when to stop/restart prefetching for its particular stream. This counter is incremented when a prefetch request is responded by the arrival of a requested line in the cache and is decremented upon its first read, i.e. this counter gives the prefetch distance; in SAHP the maximum prefetch distance is 16 cache lines. This counter and other structures needed to implement SAHP come into play at three events during execution: (1) arrival of a prefetch request at L1 cache, (2) arrival of a read request at L1 cache, and (3) when data requested through software prefetch returns to the L1 cache.

The flowcharts in Figure 3 depict how the new hardware structures are updated in SAHP at each of the three events noted above. Upon the arrival of a **Software Prefetch Request**, if the line at the start address is not in the L1 cache or MSHR file, then an MSHR is allocated that would remain bound to the prefetch request until all requested lines (denoted by nl) are prefetched. However, if the line is already present in the cache and was prefetched earlier (indicated by the prefetch bit in the line), then the current prefetch request is ignored since this denotes a possible overlap of prefetch requests (and perhaps a redundant request). If the line has not been prefetched, then the prefetch is initiated, starting at the following cache line.

When the requested line returns (i.e. **Software Prefetch Response**), the prefetch bit in the line is set, and all the structures in the MSHR are updated to initiate prefetch of the following lines. As cache lines are being prefetched continuously for a given stream, three events can happen, (a) all lines are prefetched (i.e. $nblocks$=0) and the MSHR is deallocated, (b) the prefetch distance denoted by $counter$ reaches 15 cache lines, and prefetching for this stream stops, and (c) the stream reaches the page boundary, and the physical address of the next page is retrieved from the DTLB by indexing it via the virtual address of the next page that is being tracked in the MSHR (note that this may also trigger a page walk on a DTLB miss, and SAHP thus also helps to hide this latency of a page walk).

When a prefetched block is read for the first time via a **Read Request**, the counter is decremented and the block is marked as read. If the prefetch distance denoted by $counter$ drops below 8 lines, then the stop bit is reset and prefetching is resumed. Thus, the counter and other structures help to always maintain the prefetch distance for all streams between 8 and 16 cache lines, thus preventing L1 cache pollution and also running sufficiently ahead to hide memory latency. When a read request hits in an MSHR that is servicing a prefetch request to the same cache line and is awaiting response from the lower levels of the memory, the same actions are taken to update the structures.

The num field of an MSHR that is incremented when a prefetched block is read, is indicative of the utility (i.e. how many lines prefetched by this request have been used by the program) of the prefetch request bound to the MSHR. The value of num proves handy when the MSHR Queue is full, and a new request wants an MSHR. In such a case, an MSHR servicing a prefetch request that is currently dormant (i.e. its stop bit is set) and has the lowest value of the num field is deallocated to make way for the new request. This thus helps to prevent a deadlock that may result if all prefetch requests are useless and thus come to a stop, but continue to hold the MSHR. Similarly, if a read request hits an MSHR servicing a prefetch request that is dormant (i.e. prefetch distance or counter is greater than 8; this only happens when the data stream being prefetched is irregular or discontiguous), the counter is reset and prefetching is resumed to prevent a deadlock. Finally, if a prefetch request fetches data that is already present in the L1 cache, the prefetch request is squashed given the likelihood that the request is redundant. This also helps to resolve overlapping prefetch requests.

5. EXPERIMENTAL SETUP

For our evaluation, we use the gem5 simulator [15] to model an in-order core, a full out-of-order core, and our partial out-of-order core (that implements SAHP); all cores run the ARMv7 ISA. In gem5, we simulate an in-order core by using a trimmed down version of the out-of-order core model that allows only one outstanding miss, i.e. the load/store queue has a single entry. We model a partial out-of-order core using the MinorCPU model in gem5. In the following section, we compare the performance and power of these cores armed with different prefetchers. Table 4 details the microarchitectural details of these three cores used in the simulator.

Component	Type of core		
	in-order	out-of-order	WearCore
Core		2GHz, 2-way superscalar	
Reorder logic	-	Full OoO (renaming + speculation)	Partial OoO (scoreboard)
Prefetcher	-	GHB-based stream prefetcher	SAHP
L1-Icache		32KB, 2-way LRU, 2 MSHRs, 1 cycle	
L1-Dcache		32KB, 4-way LRU, 10 MSHRs, 2 cycles	
L2 Unified	512KB	1024KB	512KB
Cache		8-way LRU, 16 MSHRs, 12 cycles	
DRAM		LPDDR3, 110ns latency [5]	
Execute units		1 int, 1 int/branch, 1 fp/simd, 1 read, 1 write	
Op latency		int: alu=1, mul=3, div=12 float: add/cmp=1, mul=3, div=12 SIMD: floatMultAcc=8, floatDiv=18, others=4 read/write=1	

Table 4: Microarchitectural details of the simulated cores

We test these cores against the ten applications in WearBench discussed in Section 2. For simulation, we identify the most representative portion of the program, i.e. that which is executed again and again during program execution and thus contributes almost entirely to the execution time. For applications that involve DNN/CNN computation (i.e. applications in the category of automatic speech recognition and computer vision), we simulate 2-4 iterations (depending on the problem size) of the outermost loop of the tiled *sgemm* kernel with respective problem sizes. We verify on real hardware that this does not alter the data reuse pattern in either the L1 or L2 cache and is thus representative of the entire kernel execution; the execution time scales linearly with the iterations of outermost loop. For the MP3 playback and H264 decoder applications, we simulate the decoding of each frame of the input audio and video, respectively, since the entire application consists of iterative decoding of all frames in the input. For the image processing applications, we simulate the entire program execution of *denoising* and *jpegc* with input images of sizes 80x80 and 800x600, respectively.

We use the GNU C/C++ Compiler to compile these applications statically for use with the simulator. We use '-funroll-loops -march=armv7 -mtune=cortex-a15 -funsafe-math-optimizations -mfpu=neon-vfpv4' options in the compiler to enable loop unrolling and vectorization wherever possible. In some cases, we manually unroll the loops to get better performance such as in the *sgemm* kernel and also in the subroutine performing forward discrete-cosine transform (DCT) in the denoising application. Finally, we use CACTI 6.5 [32] to obtain our area and power estimates for the 28nm technology node. We obtain dynamic power by calculating the total energy dissipated by the component from the per-access energy values obtained from CACTI, and then dividing it by the time taken to execute the simulated application.

Configuration	Details
InO	The baseline in-order core without any prefetchers. This core is representative of low-power ARM cores used in low-end smartphones and in some smartwatches.
SB	Partial out-of-order execution support through scoreboarding. This core allows two outstanding misses; does not use prefetchers.
O3	This core supports full out-of-order execution using register renaming and speculation. It uses an aggressive hardware prefetcher at L2 cache with a prefetch degree of 8.
SB+SAHP	This is WearCore. It uses scoreboarding and implements our proposed SAHP.
SB+SAHP-L2	Same as above. This core chooses to not have an L2 cache to represent low-power cores or cores in low-power mode.
O3+SAHP	This is an out-of-order core with SAHP, but no hardware prefetchers.

Table 5: Summary of hardware configurations tested

6. RESULTS AND DISCUSSION

Figure 4 compares the performance achieved by different hardware configurations with respect to an in-order core (without hardware prefetching) as baseline. Table 5 summarizes the different configurations; note that although we do not specifically include BIG.little, the BIG OoO core and the little InO core is included separately for comparison. The first ten groups of columns show the performance for each application in WearBench, and the last column shows the geometric mean of performance achieved by each configuration on all applications. In summary, we find that the performance of our proposed core, WearCore (denoted by SB+SAHP in the figure), gets reasonably close to the O3 core - it is 24% slower for WearBench on average. While WearCore incurs the worst performance (42% slower than *O3*) in *denoising*, it is only 12% slower for the critical *speech-to-text* application, which has strict QoS requirements. It is important to observe that *SAHP* adds a crucial 37% performance on top of a core with scoreboarding (denoted by *SB*). When using a partial out-of-order core with SAHP but without an L2 cache (i.e. *SB+SAHP-L2*), the performance degradation is 36% with respect to *O3*. Since the performance degradation is not too drastic (the performance is sometimes even slightly better in cases where there is not much reuse in L2 since we save an L2 lookup every time we go to DRAM), this is also a reasonable alternative that can be useful when the core wants to go to a low-power mode by turning off the L2 cache to conserve battery. Again, the use of SAHP makes this configuration feasible since it can prefetch the data all the way to the L1 cache. Finally, the last bar represents an O3 core with SAHP but no hardware prefetching. This core performs 5% better than *O3* because of prefetching the data to the L1 cache as opposed to the L2 cache. The performance improvement achieved by *O3+SAHP* is as large as 18% for *imc* application, thereby advocating prefetching to the L1 cache. SAHP makes this possible at a low area and power overhead as opposed to a hardware prefetcher, making its incorporation feasible in low-power devices such as a smartwatch.

We next discuss the results for application categories.

Automatic speech recognition and computer vision. The core computation in the six applications that belong to these two categories is the DNN (speech recognition) or CNN (computer vision). Among these six, the *s-to-t*, *imc* and *face* spend most of their time in NN computation (i.e. the *sgemm* kernel), and benefit more from SAHP than *t-to-c*. The reason is that SAHP benefits *sgemm* particularly because of bringing the data to the L1 cache, which is critical given that the computation is entirely SIMDized and completes quickly (thereby stressing memory). This benefit also shows when SAHP is used with *O3*. SAHP, on the other hand, is unable to benefit the computation outside NN in *t-to-c* such as the *viterbi* dynamic programming algorithm to find the most sequence of hidden states, since there are not many accesses to memory. In such cases, *SB* has to stall for every unresolved RAW dependence at the scoreboard. This makes it slower than *O3*. Also, we observe that in *imc*, SAHP adds more performance than in other applications. This is because data gets repeatedly replaced from the L1 cache due to pronounced conflicts arising from the problem size being a multiple of a power of two, and thus prefetching data to L1 boosts the performance more.

Image processing. In *denoising*, the performance gap between WearCore (i.e. *SB+SAHP*) and *O3* is maximum. This is because among all applications, the program spends the maximum fraction of time in compute. The program does read the patches obtained from the input image from the memory (during which SAHP assists), but this is followed by 16x16 DCT and inverse-DCT, where all the data used resides in the L1 cache. In *jpegc* also, the time spent in computation is significant, although SAHP improves performance when the image is read from memory.

Audio playback and video decoding. In *mp3*, SAHP adds a mild 4% improvement over *SB* only. This is because we find that the working set is only around 20KB, and so data is reused from the L1 cache. Also, the program is thus dominated by computation. In *h264dec*, on the other hand, the input frames are larger and processing the frames requires copying them, which yields sufficient opportunities to benefit from prefetching data to L1 cache through SAHP.

6.1 Area and Power Overheads

Component	Organization	Ports	Area (μm^2)	Power (mW)
MSHR file	10 entries x 4 bytes (CAM)	1 r/w 2s	1471	0.144
SAHP overhead in MSHR file (61 bits)	10 entries x 61 bits	1 r/w	1865	0.005
SAHP overhead in Cache (6 bits)	512 blocks x 6 bits	1 r/w	2114	0.034
Input Buffer	16 entries x 4 bytes (CAM)	1 r/w 2s	1471	0.144
Scoreboard	64 entries x 1 byte	1r 1w	2956	0.134
InFlightInstQueue	16 entries x 4 bytes (CAM)	1 r/w 2s	1471	0.144
Store Buffer	8 entries x 4 bytes (CAM)	1 r/w 1s	931	0.01
WearCore			462,279	100.5
Cortex-A15			2,025,000	450

Table 6: Area and power overhead of WearCore over an in-order ARM Cortex-A7 core (Area=450,000μm^2, Power=100mW) when modeled with CACTI6.5 in 28nm. The area and power consumed by ARM Cortex-A15 core is also listed. The numbers do not include the L2 cache.

For the purpose of determining area and power requirements of WearCore, we follow the strategy as adopted in [17] - we use the area and power numbers of a baseline in-order core (ARM Cortex-A7 in our case) as available publicly, and calculate the overhead of the components added over it to implement WearCore. According to ARM [2], the core area and average power consumption of Cortex-A7 are 0.45 mm^2 and 100mW, respectively, in 28nm. WearCore and Cortex-A7 are also compared against the out-of-order ARM Cortex-A15 core, whose area and power consumption are 4-5 times larger than Cortex-A7 [21, 1].

Table 6 lists for each major component in WearCore, the area and average power consumption as calculated using CACTI v6.5 [32] at 28nm. Note that whenever we require to round the size of the structure to the nearest power of 2 in CACTI, we round it to the larger number. The overheads stem from the following causes. (1) In order to support SAHP, WearCore uses a larger MSHR file at the L1 cache as opposed to A7 that only supports two outstanding prefetch requests through the PLD instruction (and one outstanding load). We modeled this component as a CAM. (2) The 61 bits in each MSHR in order to keep track of the prefetch distance in hardware. We model this component as a direct-mapped cache since the corresponding entry is accessed directly either through the *mID* field in a cache block during a cache hit or when the prefetch returns. (3) The 6 bits in the cache. This is modeled as in (2). (4) The input buffer buffers the instructions before the scoreboard determines that it is safe to issue them. We model this as a CAM since loads in this buffer may bypass other instructions thereby requiring an associative lookup. (5) The scoreboard simply tracks the number of in-flight instructions writing to any particular register. It allows a new instruction to issue once there is no in-flight instruction waiting to write to any of the source registers of the new instruction (indicated by a zero count of the source registers in the scoreboard). We model this as a direct-mapped cache since it is indexed by an instruction's source register. (6) The InFlightInstQueue is responsible for in-order commit of all instructions. It stores all instructions that have been issued but have not been completed. (7) The store buffer is used to forward the data to cacheable loads and has 8 entries in our case. In total, the area and power overhead of WearCore is merely 2.7% and 0.5%, respectively, compared to the baseline.

We combine these area and power estimates with the perfor-

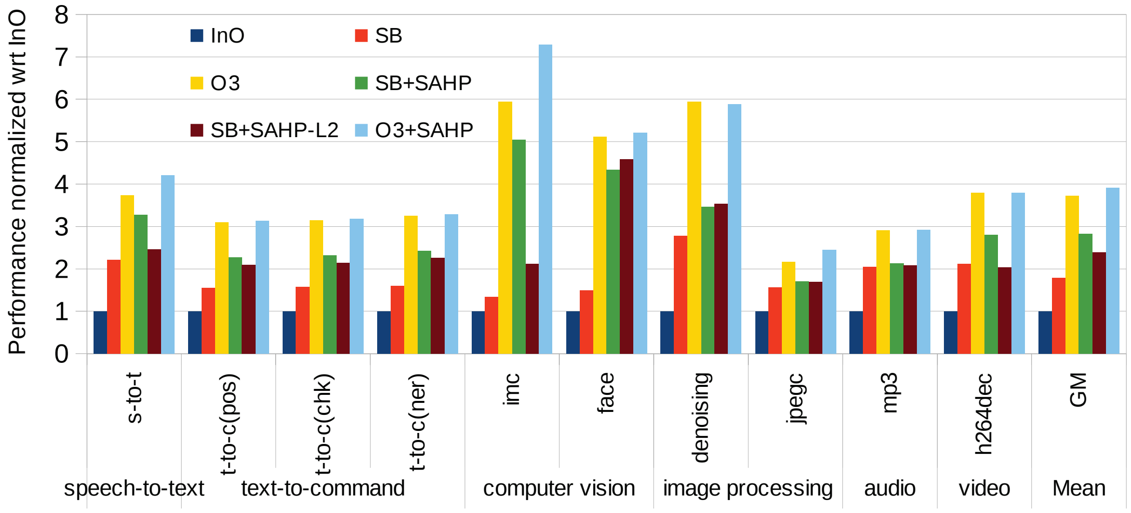

Figure 4: Performance comparison of different cores with their respective prefetchers

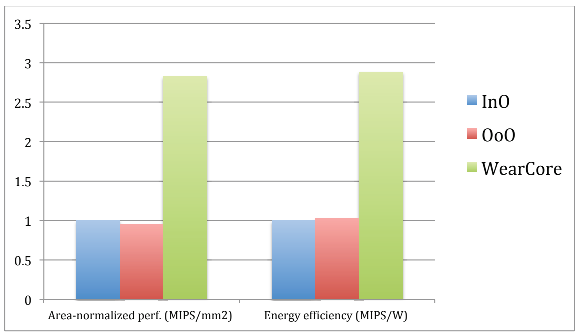

Figure 5: Area-normalized performance and energy-efficiency of WearCore versus an in-order and an out-of-order core

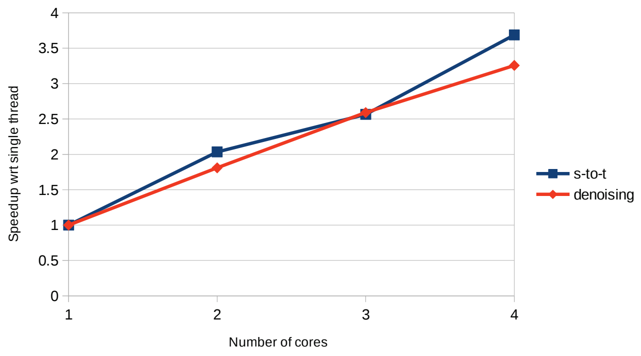

Figure 6: Parallel performance of two WearBench applications; the dip in *s-to-t* at 3-cores is due to load imbalance

mance numbers obtained from gem5. As shown in Figure 5, WearCore achieves an improvement of 2.8x and 2.9x respectively, in terms of area-normalized performance and energy efficiency over an in-order core. On the other hand, the in-order core achieves very similar area-normalized performance and energy efficiency as an out-of-order core. This result is similar to published data [21] about ARM's Cortex-A7 and A15 CPUs. Note that these numbers include the area and power of the L2 cache in each core for better estimates, and that the out-of-order core uses an L2 cache that is double the size of L2 in the in-order core as well as WearCore.

6.2 Parallel Performance

Figure 6 shows the parallel performance achieved by two applications from WearBench, *speech-to-text* and *denoising*. We chose the former as a representative of all six applications that spend the majority of their time in neural network computation and also because this application has strict QoS requirements. The figure shows that these applications scale well with the number of cores, and thus prove the utility of employing multiple cores on the chip. *Speech-to-text* benefits particularly since it can now achieve its target QoS - using four cores, we find that the DNN computation involved in recognizing 5 seconds of speech completes in under 250ms (as opposed to nearly a second on a single core) on WearCore when simulated on gem5 in full-system mode. Also, an 80x80 image can be denoised in 3.5 seconds as opposed to 11.5 seconds.

6.3 Discussion

We thus see that applications in WearBench benefit considerably from WearCore, our proposed core that extends a baseline partial out-of-order core with a software-assisted hardware prefetcher. This proves to be useful in WearBench because of reducing stalls on memory accesses while the computation completes quickly as a result of vectorization. This benefit, however, requires the insertion of our proposed software prefetch instruction in the code by library developers and users. Library developers currently tune their libraries to benefit from the SIMD extensions on the host hardware, such as NEON for ARM. This approach has been tenable for smartphones till now, particularly because of the select applications that run on them. For example, the libraries used in audio and video encode/decode (such as libavcodec) currently provide specific implementations that benefit from the SIMD extensions. This strategy is also adopted by smartphone companies such as Intel, which recently released its chip for smartphones that can perform offline speech recognition in an efficient way by software-hardware co-development [6].

We thus believe that these libraries could similarly include extensions for prefetching in the source code to benefit from SAHP. Although such software prefetch instructions exist in current ARM ISA, but is not beneficial to insert them because of the associated overhead and due to the difficulty in determining where to insert them and what prefetch distance to use. The prefetch instruction in SAHP overcomes both of these limitations, making the task of library developers and users much simpler.

7. RELATED WORK

In the present day, a key goal of computer architects is to extract the maximum energy efficiency from the core. This goal is only getting renewed importance as consumers move from desktop PCs to laptops to smartphones, and now to smartwatches. As architects contend with this goal, they face the well-known tradeoff between performance and power. In order to gain performance, some power-consuming component must be added on the chip. Thus, the ultimate gain in terms of energy-efficiency depends on whether the target applications benefit from the newly added component or not. As we expect the market share of smartwatches to improve, it is important to identify the key target applications and then propose minimal hardware to improve its energy-efficiency.

Workload Characterization. In the recent past, there has been considerable research on identifying key target applications for the smartphone. BBench [26] and MobileBench [39] are representative examples. However, these works have not considered on-device speech recognition or face recognition using DNNs, which are of particular importance to a smartwatch. More recently, Gao et al. [23] show that mobile applications utilize less than 2 cores on average, and thus most mobile processors are overprovisioned with cores with some of them now having as many as eight cores on the chip. We find that having multiple cores of the chip in a smartwatch can be particularly useful for a critical application like *speech-to-text* that has strict quality of service requirement; other applications also make use of multiple cores. We therefore include these workloads in WearBench and propose to have multiple simple cores on chip in a smartwatch. We also show that these applications benefit a lot from the SIMD unit on the core, making for a case to improve memory performance.

In order to improve memory performance, there have been various solutions proposed, mostly hovering around prefetching data to the higher levels of memory hierarchy in order to prevent miss latency.

Software-only solutions to improve energy-efficiency. Software prefetching [16, 37] is a well known technique to hide memory latency. It is for this reason that all existing cores including some of the in-order ARM cores provide support for software prefetching. However, as discussed in Section 3, using software prefetch instructions does not necessarily benefit performance due to the overhead associated with introducing them in the innermost loop. In this work, we use a software prefetch instruction to trigger prefetching, but that instruction appears outside the innermost loop to prevent this overhead.

Software-hardware solutions. Recently, Mehta et al. [35, 22] have proposed coordinated prefetching that relies on coordination between hardware prefetcher and software prefetching. They show that the overhead due to software prefetching can be overcome on Intel's in-order Xeon Phi cores, but they rely on the fact that the vector width is the same as the cache line size. Thus, in that case, there are no redundant prefetch instructions, i.e. there is one prefetch instruction per cache line provided the innermost loop is vectorized. However, the vector width on ARM cores is a fourth of the cache line size and thus redundant prefetches cannot be avoided even when the innermost loop is vectorized. Similarly, the idea of block prefetching where a single prefetch instruction triggers requests to n (usually 4) consecutive lines, suffers from the problem of redundant prefetches; elimination of redundant prefetches via conditional-code/unrolling/short-loops (from strip-mining) prove detrimental to vectorization performance and is thus not used in current production compilers. Other research [29, 41, 28] employs helper threads to predict future load addresses. The authors in [29, 28] use an idle thread (SMT) as the helper thread to prefetch data for another compute thread, whereas Son et al. [41] extend the helper-thread prefetching to work with multiple (many) cores by assigning a customized helper thread to a group of compute threads. However, these solutions are less relevant for smartwatches that do not employ SMT (to keep them simpler) and have many fewer cores than those in current multi- (many-) cores.

Hardware-only solutions. An example in this category is runahead execution [38] where execution does not stop at a cache miss resulting in future data being prefetched to the cache in time. Other examples include designs where the data is precomputed before it is needed [19] and those where the processor resources are freed up for miss-independent instructions [42]. However, these solutions are all proposed for out-of-order cores and require sufficient hardware themselves. These are thus not well suited to low-power devices. Recently, Carlson et al. [17] have proposed the Load Slice Core Microarchitecture that extracts memory hierarchy parallelism (MHP) by enabling memory accesses along with their address-generating instructions to execute while the pipeline is stalled on a long-latency miss. They propose a separate pipeline for independent memory accesses (and their address-generating instructions), and additional hardware that identifies address-generating instructions that lead up to the independent memory accesses. They achieve good energy efficiency by extracting MHP over both in-order and out-of-order cores. While their technique is more general and targeted towards many-core processors, we achieve similar gains for applications in WearBench with simpler hardware, and WearCore is therefore better suited to smartwatches. Another solution in this category that comes close to our work is Guided Region Prefetching (GRP) [45]. In GRP, the compiler provides a hint to the hardware prefetcher about the loop trip count, and the hardware then determines all blocks to be prefetched in a page (region) and dynamically tracks all remaining blocks and the next block to be prefetched in each page. However, since this is done for each stream, and GRP may prefetch arbitrarily ahead on the page (4 KB in its case) for each stream, it starts displacing even the prefetched data before being used in the event of multiple data streams. The key in SAHP, on the other hand, is that the hardware maintains a counter that tracks the current prefetch distance, and the hardware never allows itself to prefetch too far ahead to start displacing useful/prefetched data. GRP also requires re-triggering of the prefetcher (and therefore additional memory accesses) at every new page, while SAHP goes across pages without stopping. Overall, since SAHP requires much less additional hardware, it is more suitable for a smartwatch.

8. CONCLUSION AND FUTURE WORK

In this work, we propose WearCore, a core for efficiently executing smartwatch workloads, that we term WearBench. WearCore is built on the insight that the important workloads for a smartwatch are parallel, and make extensive use of SIMD operations, making it important that the memory requests be serviced quickly for overall good application performance. Thus, building upon a quad-core processor as baseline, WearCore augments it with a light-weight software-assisted hardware prefetcher for reducing the cache misses altogether. This software-assisted hardware prefetcher adds minimal overhead in terms of area and power to the core, but adds a crucial 36% performance on top of the baseline. It is also very easy to use, and does not add instruction overhead either. Experimental results show that WearBench achieves significant improvement in terms of both area and energy efficiency over both an in-order and an out-of-order core.

Acknowledgement This work was supported in part by NSF under grants CCF-1012759 and CCF-1536795.

9. REFERENCES

[1] "Arm cortex-a15," Available at http://www.arm.com/products/processors/cortex-a/cortex-a15.php.

[2] "Arm cortex-a7," Available at http://www.arm.com/products/processors/cortex-a/cortex-a7.php.

[3] "Arm cortex-a9," Available at https://www.arm.com/products/processors/cortex-a/cortex-a9.php.

[4] "Arm streamline performance analyzer," Available at http://ds.arm.com/ds-5/optimize/.

[5] "Cortex a15 dram latency," Available at http://www.7-cpu.com/cpu/Cortex-A15.html.

[6] "Intel voice recognition will blow siri out of the water because it does not use the cloud," Available at http://qz.com/170668/intels-voice-recognition-will-blow-siri-out-of-the-water-because-it-doesnt-use-the-cloud/.

[7] "mpeg3play mp3 music player," Available at http://www.mp3-tech.org/programmer/sources/mpeg3play-0_9_6-src.tgz.

[8] "Openblas - an optimized blas library," Available at http://www.openblas.net/.

[9] "Samsung galaxy s6: Battery life," Available at http://www.trustedreviews.com/samsung-galaxy-s6-review-battery-life-and-charging-page-4.

[10] "Samsung gear s 3g smartwatch: Battery life," Available at http://www.extremetech.com/computing/188828-samsung-unveils-standalone-gear-s-3g-smartwatch-awesome-until-the-battery-runs-out-after-an-hour.

[11] "Logging into twitter and facebook using your face," 2010, Available at http://thenextweb.com/mobile/2010/11/10/brilliant-logging-into-twitter-and-facebook-using-your-face-and-voice-video/.

[12] "Can a smartwatch like the gear s replace your phone?" 2015, Available at http://www.techradar.com/us/news/wearables/can-a-smartwatch-replace-your-phone--1285484.

[13] "The problem of typing on a watch," 2015, Available at http://www.cnet.com/products/samsung-gear-s/.

[14] ABI-Research, "Wearable computing devices, like apple iwatch, will exceed 485 million annual shipments by 2018," 2013, Available at https://www.abiresearch.com/press/wearable-computing-devices-like-apples-iwatch-will/.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[16] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 40–52.

[17] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 272–284.

[18] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, "Hexagon dsp: An architecture optimized for mobile multimedia and communications," *IEEE Micro*, vol. 34, no. 2, pp. 34–43, Mar 2014.

[19] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 34. Washington, DC, USA: IEEE Computer Society, 2001, pp. 306–317.

[20] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12, pp. 2493–2537, Nov. 2011.

[21] EE-Times, "How arm's cortex-a7 beats the a15," 2013, Available at http://www.eetimes.com/author.asp?section_id=36&doc_id=1318968.

[22] Z. Fang, S. Mehta, P.-C. Yew, A. Zhai, J. Greensky, G. Beeraka, and B. Zang, "Measuring microarchitectural details of multi- and many-core memory systems through microbenchmarking," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 55:1–55:26, Jan. 2015.

[23] C. Gao, A. Gutierrez, M. Rajan, R. G. Dreslinski, T. Mudge, and C.-J. Wu, "A study of mobile device utilization," *2015 IEEE International Symposium on Performance Analysis of Systems and Software*, 2015.

[24] Google-Research, "Voice search made faster," 2015, Available at http://googleresearch.blogspot.in/2015/09/google-voice-search-faster-and-more.html.

[25] Google-Wear, "Android wear on wi-fi: Using a smartwatch without a phone nearby," 2015, Available at http://www.computerworld.com/article/2919013/android/android-wear-on-wi-fi-using-a-smartwatch-without-a-phone-nearby.html.

[26] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-System Analysis and Characterization of Interactive Smartphone Applications," in *the proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, USA, 2011, pp. 81–90.

[27] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, R. Dreslinski, T. Mudge, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, ser. ISCA '15. New York, NY, USA: ACM, 2015.

[28] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 393–404.

[29] D. Kim and D. Yeung, "Design and evaluation of compiler algorithms for pre-execution," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 159–170.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[31] X. Lei, A. Senior, A. Gruenstein, and J. Sorensen, "Accurate and compact large vocabulary speech recognition on mobile devices," in *Proceedings of the 14th Annual Conference of the International Speech Communication Association*, ser. ISCA '13, 2013.

[32] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Proceedings of the International Conference on*

Computer-Aided Design, ser. ICCAD '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 694–701.

[33] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," 2016.

[34] S. Mehta, G. Beeraka, and P.-C. Yew, "Tile size selection revisited," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 35:1–35:27, Dec. 2013.

[35] S. Mehta, Z. Fang, A. Zhai, and P.-C. Yew, "Multi-stage coordinated prefetching for present-day processors," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 73–82.

[36] S. Mehta, R. Garg, N. Trivedi, and P.-C. Yew, "Turbotiling: Leveraging prefetching to boost performance of tiled codes," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 38:1–38:12.

[37] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS V. New York, NY, USA: ACM, 1992, pp. 62–73.

[38] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, ser. HPCA '03, Washington, DC, USA, 2003.

[39] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite - mobilebench," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, Sept 2013, pp. 133–142.

[40] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, "The kaldi speech recognition toolkit," in *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, 2011.

[41] S. W. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti, "A compiler-directed data prefetching scheme for chip multiprocessors," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 209–218. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504208

[42] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. New York, NY, USA: ACM, 2004, pp. 107–119.

[43] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*. IEEE, 2014, pp. 1701–1708.

[44] The-Guardian, "Why i have finally taken off the apple watch for the last time," 2016, Available at https://www.theguardian.com/technology/2016/jun/09/apple-watch-smartwatch.

[45] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided region prefetching: A cooperative hardware/software approach," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA '03. New York, NY, USA: ACM, 2003, pp. 388–398.

[46] G. Yu and G. Sapiro, "Dct image denoising: a simple and effective image denoising algorithm," *Image Processing On Line*, vol. 108, 2011.