Processor Customization for Wearable Bio-monitoring Platforms

Huynh Phung Huynh Tulika Mitra Department of Computer Science National University of Singapore {huynhph1, tulika}@comp.nus.edu.sg

Abstract—Wearable bio-monitoring applications require significant computation bandwidth. Processor customization is a major technology trend that can potentially satisfy computation requirement. In this paper, we apply processor customization to wearable bio-monitoring platforms and create systems with high performance. We choose Stretch customizable processor as the hardware platform where application-specific extension instructions are implemented in reconfigurable logic. Experimental results demonstrate that processor customization can return a performance gain of up to 5.2X.

I. INTRODUCTION

The increasingly ageing population is posing a major challenge to the overall health-care systems worldwide. Remote and non-obtrusive continuous bio-monitoring of a non-critical patient at home is a viable alternative that can reduce considerable burden on the hospital resources. Wireless body-area sensor networks (or BANs) and related wearable computing technologies promises a convenient platform for such biomonitoring applications. The recent technological advancements in embedded processors, availability of ultra low-power and lightweight sensor nodes and advances in wireless networking have all paved the ways for wireless BAN platforms.

Figure 1 illustrates the typical architecture of a wearable bio-monitoring platform. Multiple tiny sensor nodes are attached to the different parts of the patient's body. These sensor nodes continuously sample various vital signs, such as ECG (Electrocardiograph), SpO2 (Saturation of Arterial Oxygen) etc., at regular intervals and transmit the collected samples to a gateway device (typically mobile phone or personal digital assistant (PDA)) through wireless medium. The gateway device is also located in the vicinity of the person being monitored such as on his/her body. The sensor nodes communicate with the gateway device through wireless communication protocol such as ZigBee (802.15.4) or Bluetooth (802.15.1). The gateway device is responsible for processing the sampled data streams and detecting emergency conditions (such as a fall) or anomaly in the vital signs. It can employ mobile telephone networks (GPRS, 3G, etc.) or wireless LAN to reach an Internet access point and thereby trigger an alarm to the care-giver in case of an emergency or anomaly. It also periodically reports the status of the patient to the medical servers.

Clearly, the high-end bio-monitoring applications demand significant computation bandwidth from the gateway device, typically a PDA or smart phone. This is in addition to the computation bandwidth required for running regular applications on the device, such as phone calls or music players.



Fig. 1. Wearable bio-monitoring.

On the other hand, given the small form factor and battery life restrictions, the PDAs include very lightweight processors running at 100-300 MHz. Thus, there is an increasing trend towards building customized gateway devices specifically tailored towards wearable bio-monitoring platforms. As an example, recently an application-specific multiprocessor system-on-chip (MPSoC) design has been proposed for real-time analysis of a 12-lead ECG [5], which requires processing of twelve different signals from the patient's body.

Following this line of development, we focus on *processor customization* [4] to support the computation demand placed on the gateway device by high-end bio-monitoring applications. Processor customization has recently emerged as a major paradigm shift to provide scalable compute power in a short time-to-market window. A customizable processor is, in general, configurable w.r.t. the micro-architectural parameters, such as cache configurations. More importantly, a customizable processor may support application-specific extensions of the core instruction-set. Custom instructions encapsulate the frequently occurring computation patterns in an application. They are implemented as custom functional units (CFU) in the datapath of the existing processor core. CFUs improve performance and power through parallelization and chaining of operations.

In this work, we choose Stretch customizable processor [1] as the hardware platform. Figure 2 shows the Stretch S5 engine that incorporates Tensilica Xtensa RISC processor [3] and the Stretch Instruction Set Extension Fabric (ISEF). The ISEF is software-configurable datapath based on programmable logic. This configurable fabric acts as a functional unit to the processor. It is built into the processor's datapath, and resides alongside other traditional functional units such as the ALU and the floating point unit. The programmer defined application specific instructions (Extension Instructions) are



Fig. 2. Stretch S5530 datapath [1].

implemented in this fabric. The core processor issues the extension instructions to ISEF, which performs the computation and returns the result. In summary, the ISEF allows the system designers to define new instructions post-silicon and thus extend the processor's instruction-set.

The major obstacle to customization of bio-monitoring applications is that Stretch extensible processor (like many other extensible processors) does not support floating point operations within extension instructions. Unfortunately, profiling of bio-monitoring applications indicate that all the computeintensive kernels contain significant amount of floating point arithmetic operations. Therefore, we first transform the applications to use fixed point arithmetic instead of floating point arithmetic. This transformation enables better exploitation of instruction-set customization. We then generate multiple customization options for each compute-intensive kernel with varying area and performance gain. It is obvious that a customization option with larger area will typically provide better performance gain. Finally, We select appropriate customization options for all the kernels so as to obtain best tradeoff between area and performance gain for the entire application.

II. WEARABLE BIO-MONITORING APPLICATIONS

In this work, we choose a concrete bio-monitoring application from the geriatric care domain as a case study. The application consists of two related subsystems: (1) continuous monitoring of vital signs and (2) fall detection.

A. Continuous Monitoring of Vital Signs

The subsystem for monitoring vital signs is capable of continuously measuring ECG, SpO2, systolic blood pressure, and heart beat rate. ECG electrodes are attached on the chest to measure the cardiac activities. SpO2 probe irradiates red and infrared light onto earlobe and then records the continuous changes of transmitted intensities, which is called PPG (Photo Plethysmogram). In each cardiac cycle, the ECG R peak indicates the starting of cardiac contraction, and the corresponding maximum inclination in the PPG indicates the arrival of blood at earlobe. The interval between the two kinds of peaks is defined as pulse transit time (PTT) [2] as illustrated in Figure 3(a). That is, PTT is the time it takes for the blood flow to reach from the heart to the earlobe. These vital signs are continuously monitored and transmitted to the gateway device.

The continuous measurement of vital signs requires a realtime systolic blood pressure estimation algorithm as shown in Figure 3(b). The detection of PTT involves peak detection in both ECG and differentiated PPG. An Analog to Digital converter samples the ECG signal. The sampled ECG waveform contains some amount of superimposed line-frequency content. This line-frequency noise is removed by digitally filtering the samples through a low-pass FIR filter. This is followed by detection of all the QRS complex in the ECG waveform. The ECG R peaks can be easily derived from the QRS complex. The QRS complex also serves as a definite indicator for every heart beat, hence, it can be used to calculate the heartbeat rate. The PPG signal similarly goes through a FIR filter to remove the noise followed by detection of all the maximum slopes of the PPG. After R peaks of ECG and maximum slopes of PPG are detected, the corresponding pairs are mapped together to compute PTT. Finally, several PTT readings in a time interval are combined together into one blood pressure index.

B. Fall Detection

Any wearable fall detection system typically employs physical motion sensors such as tri-axial accelerometers and gyroscopes. The fall detection system we examine for case study consists of one tri-axial (3D) MEMS accelerometer plus one gyroscope on the thigh position and another accelerometer on the waist position. The sensitivity axes of each accelerometer is arranged in lateral, vertical, and antero posterior directions. The gyroscope provides 2D angular (lateral and sagittal) motion information. Overall we have eight streams of sensor signals coming in from the physical motion sensors (lateral, vertical, antero-posterior for each accelerometer and lateral, sagittal for gyroscope) to the gateway device through ZigBee (802.15.4) wireless communication protocol. The fall detection algorithm runs on the gateway device.

The central hypothesis of elderly fall detection approach is that the thigh motion does not go beyond certain threshold angle to forward (lateral) and sideways (sagittal) directions in normal activities; the abnormal behavior occurs in the onset of falls among the elderly. Moreover, there is a high correlation between thigh and waist angle during fall, but low correlation during normal activities. Thus the algorithm first needs to transform the 3D accelerometer data to 2D angular data (lateral and sagittal). Next, it marks an angular motion of the thigh beyond a threshold as a "possible" onset of fall. For each such possible onset of fall, the correlation between thigh and waist angles as well as pattern matching of gyroscope angle (against reference values obtained from a number of actual falls) are used to eliminate false positives. A high-level overview of the functionalities of the fall detection application appears in Figure 3(c).

III. PROCESSOR CUSTOMIZATION

A quick profiling of the fall detection application revealed the floating point arithmetic operations as the main performance bottleneck. Table I shows the compute-intensive functions in the fall detection application along with the percentage







(b) Blood pressure estimation.

(c) Fall detection algorithm.

Fig. 3. Bio-monitoring Applications.

Function	Execution Time	Floating Point
muldf3	46.05%	Yes
ieee754_sqrt	8.96%	Yes
divsf3	8.55%	Yes
adddf3	7.65%	Yes
filter	6.56%	No
atan	4.52%	Yes
main	3.99%	Yes
divdf3	2.61%	Yes
subdf3	2.22%	Yes

TABLE I

HOT FUNCTIONS IN FALL DETECTION APPLICATION.

of execution time spent in those functions. Most of these functions are implementations of floating-point arithmetic operations. In fact, more than 80% of the execution time of the application is spent in floating point arithmetic operations.

More importantly, the instruction-set extensible processor that we are targeting (i.e., Stretch) does not support floating point arithmetic operations within custom instructions. Indeed, most customizable processors do not support floating-point operations inside custom instructions. Consequently, we get at most 1.04x speedup after we generate Stretch custom instructions for fall detection application. Therefore, we first transform the fall detection application code to use fixed point arithmetic instead of floating point enabling better exploitation of instruction-set customization.

On the other hand, blood pressure estimation application mostly uses integer arithmetic. So, we do not need to implement fixed point arithmetic version for the blood pressure estimation algorithm.

A. Conversion to Fixed Point Arithmetic

We use N-bit binary number $x = x_{N-1}x_{N-2} \dots x_1x_0$ to present a fixed-point number in the form U(a, b) [7].

$$x = \frac{1}{2^b} \sum_{n=0}^{N-1} 2^n x_n$$
 and $a = N - b$

In this representation, a bits on the left correspond to the integer part while b bits on the right correspond to the fractional part. The implied binary point exists between the b^{th} bit x_b and the bit to its right x_{b-1} . The accuracy of the fixed point representation and the results of the corresponding arithmetic operations (compared to the floating point implementation) crucially depend on the appropriate choice of values for aand b. Therefore, we select different values of a and b for different functions depending on the accuracy requirements in our fixed point implementation of the applications. Moreover, we choose N = 32 for most of functions and N = 64 for certain functions. For our application, N = 64 is large enough to maintain the accuracy of floating-point operations when we convert them to fixed-point representation.

We convert each rational number or integer number to fixed-point representation by multiplying it with 2^b , where the value of b is chosen to maintain the appropriate accuracy. A fixed-point representation can be treated as an integer number except that it has the implied binary point separating integer and fractional parts. Therefore, if we ensure that two fixed point operands of an operation (such as addition or division) have the same values for a and b, we can use the normal integer arithmetic operations for fixed-point numbers. Stretch extension instruction can support integer multiplication, subtraction and addition operations. Therefore, we have to implement integer and fixed-point division operation using basic arithmetic operations (such as shift, or, etc.) [6].

B. Stretch Custom Instructions

A single custom instruction in Stretch can specify a complete inner loop in the application. The developer needs to capture the inner loops as extension instructions in Stretch C, which is a variant of standard ANSI-C language. The Stretch C compiler then fully unrolls any loop with constant iteration counts. There are three main sources of performance gain from the custom instructions in Stretch [1]: (1) Each custom instruction can read up to three 128-bit operands and produce up to two 128-bit operands. This allows a custom instruction to exploit significant data parallelism as multiple data values can be packed together in a single 128-bit operand. (2) A custom instruction can exploit temporal parallelism through a deeply pipelined implementation of up to 27 processor clock cycles. (3) Each custom instruction can be specialized through bit width optimization, constant folding, partial evaluation, and resource sharing.

Now how do we specify and use custom instructions in Stretch to achieve performance gain for our application?

Figure 4 shows an example of exploiting custom instructions on Stretch processor. The original source code is shown as Figure 4(a). It performs FIR filtering on the 16-bit elements



Fig. 4. An example of custom instruction for Stretch processor.

in the buffer *buf*. The custom instruction, called CI_filter (Figure 4(b)), has two WRs, A and B, and an unsigned short *offset_8* as arguments. Each of A and B contains eight input elements that will be multiplied and accumulated. First, input data in A and B are unpacked to the local variables *input0* and *input1*. Then *input0* and *input1* are multiplied and accumulated to z. The Stretch C compiler, while synthesizing the custom instruction into hardware, will unroll this for loop within the custom instruction. Finally, z is packed into A register as the output. After the new custom instruction is defined, we have to change the source code of the original loop to use the newly defined custom instruction (see Figure 4(c)).

IV. EXPERIMENTAL RESULTS

We write Stretch C instructions for each hot function to explore speed up of bio-monitoring application. Then we used Stretch profiler to get cycle count of each function in the biomonitoring application. Moreover, after generating bit stream configuration of custom instructions, we get the hardware area (in terms of number of arithmetic/logic units (AU) and multiplier units (MU)) of each custom instruction for each hot function. Different combinations of custom instructions create different custom instruction-set versions for each hot functions.

From custom instruction-set versions generated for hot functions, we choose appropriate custom instruction-set version for each hot function of the bio-monitoring applications. We vary the hardware area constraint from 0 to Max_Area at a hardware unit of 0.1 x Max_Area. The Max_Area is simply the summation of the maximum hardware area requirements of the constituent bio-monitoring kernels. Bio-monitoring application enhanced with custom instructions at Max_Area explores the limit of speedup achievable. In Figure 5, the X-axis and Yaxis represent area constraints and speedup of the application respectively. Recall that blood pressure estimation application mostly uses integer arithmetic. Therefore, we only enhance blood pressure estimation application with custom instructions and we can get up to 1.5x speedup shown in green bar in Figure 5, bp_sw_custom. Here, the speedup is the ratio of blood pressure application execution time in software to the



execution time (in cycles) of the application enhanced with custom instructions.

On the other hand, we have three implementations for fall detection application: (1) software-fixed-point implements fixed point arithmetic in software. (2) software-floating-point implements floating point arithmetic in software. (3) customfixed-point implements fixed point arithmetic with custom instructions. The custom-fixed-point implementation gets up to 1.97x speedup (blue bar in Figure 5) compared to the softwarefixed-point implementation, fd_sw_fixed_custom_fixed. Performance speedup also comes from the fixed point arithmetic implementation instead of the floating point implementation. Red bar in Figure 5 shows final speedup of customfixed-point implementation over software-floating-point one, fd_sw_float_custom_fixed. We can get nearly 5.2x performance speedup compared to the original floating-point implementation of fall detection application while the accuracy of arithmetic operations is still maintained.

V. CONCLUSIONS

In this paper, we present our work on processor customization for bio-monitoring applications. Our customization is based on fixed point implementation and custom instruction selection. Through customization, we can get high performance gain (5.2x). The result of this work confirms the efficiency of processor customization for compute-intensive application domains such as bio-monitoring applications.

VI. ACKNOWLEDGMENTS

This work is supported by A*STAR SERC project R-252-000-258-305. We would like to thank Francis Eng Hock Tay and Nyan Myo Naing for sharing the bio-monitoring application with us.

References

- J. M. Arnold. S5: The Architecture and Development Flow of a Software Configurable Processor. In *ICFPT '05*.
 P. Fung, G. Dumont, C. Ries, C. Mott, and M. Ansermino.
- [2] P. Fung, G. Dumont, C. Ries, C. Mott, and M. Ansermino. Continuous Noninvasive Blood Pressure Measurement by Pulse Transit Time. In *EMBS '04*.
 [3] R. E. Gonzalez. Xtensa: A Configurable and Extensible Proces-
- [3] R. E. Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2), 2000.
 [4] P. Ienne and R. Leupers, editors. *Customizable Embedded*
- [4] P. Ienne and R. Leupers, editors. Customizable Embedded Processors: Design Technologies and Applications. Systems on Silicon. Morgan Kaufmann, 2006.
 [5] I. A. Khatib, D. Bertozzi, A. Jantsch, and L. Benini. Performance
- [5] I. A. Khatib, D. Bertozzi, A. Jantsch, and L. Benini. Performance Analysis and Design Space Exploration for High-end Biomedical Applications: Challenges and Solutions. In CODES+ISSS '07
- Applications: Challenges and Solutions. In CODES+ISSS '07.
 [6] D. A. Patterson and J. L. Hennessy. Computer Organization & Design. Morgan Kauffman, 1998.
 [7] R. Yates. Fixed-point Arithmetic: An Introduction. Technical
- [7] R. Yates. Fixed-point Arithmetic: An Introduction. Technical report, Digital Signal Labs, 2007.