A Fault-Tolerant Hardware Architecture for Robust Wearable Heart Rate Monitoring

Qingkun Li, Homa Alemzadeh, Zbigniew Kalbarczyk, Ravishankar K. Iyer
Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL, USA
{qli19, alemzad1, kalbarcz, rkiyer}@illinois.edu

***Abstract*—This paper presents a fault-tolerant hardware architecture for robust wearable heart rate monitoring. The proposed architecture is designed for fusion of the heart rates estimated from both electrocardiogram (ECG) and arterial blood pressure (ABP) signals, with small hardware footprint and low energy consumption. It benefits from the following unique features: (1) an optimized heart beat (peak) detection algorithm that can be dynamically configured for either ECG or ABP analysis, resulting in about 38% reduction of the hardware footprint, (2) coarse-grained reconfigurable functional units (FUs) that can be programmed for different processing flows, and (3) a low overhead fault detection and recovery unit that enables dynamic recovery from transient hardware faults in the FUs. Both FPGA and ASIC prototypes of the proposed hardware have achieved much better performance and energy efficiency compared to an Android implementation of the same algorithm, and can recover from transient faults with low resource (~15%) and energy (~34%) overheads and no (0%) performance impact.**

Keywords—heart rate monitor; reconfigurable architectures; fault tolerance; biomedical monitoring; wearable monitoring

I. INTRODUCTION

Cardiac arrhythmias affect more than 5 million Americans, resulting in more than 1.2 million hospitalizations and 400,000 deaths each year in the U.S. [1]. Arrhythmias are often characterized by suddenness and unpredictability, and may not be effectively detected by regular examinations. Therefore, pervasive real-time heart rate monitoring is key to in-time detection of arrhythmia and prevention of severe consequences.

A wearable heart rate monitoring system requires three main factors in order to provide robust and continuous real-time monitoring: (1) accuracy, (2) portability, and (3) long battery life. Accuracy depends on: (1) correctness and quality of input signals collected from sensors, (2) adequacy of the signal processing algorithms, and (3) reliability of the underlying hardware that runs the monitoring algorithms [2]. However, the existing wearable monitors suffer from numerous false alarms and delayed feedback, due to signal noise, artifacts [3], and missing data (e.g., due to disconnections) [4]. Also, a study of previous recalls of monitoring systems shows that system malfunctions due to software and hardware failures can cause serious safety problems, sometimes even resulting in patient deaths [5]. Therefore, there is a compelling need for design of reliable systems for pervasive heart rate monitoring.

Several studies have proposed noise filtering and signal quality assessment techniques [6]-[8] or accurate signal peak detection algorithms [9][10], to address the problem of false alarms due to erroneous heart rate estimations. Others have proposed sensor fusion techniques to provide robust heart rate estimations by obtaining and fusing information from multiple physiological signals, such as the electrocardiogram (ECG), arterial blood pressure (ABP), and pulse oximetry waveforms [4][11][12][13]. Among those, Kalman-filter-based sensor fusion has been the most reliable technique for estimating heart rate from multiple noisy signals that provide redundant and independent measures of heart rate [4][12].

However, the implementation of sensor fusion algorithms on wearable devices requires concurrent recording, analysis, and/or transmission of multiple biomedical signals collected at relatively high sampling rates (125-360 Hz) from the patient's body. But with the portability and long battery life requirements, wearable monitoring devices face challenges in real-time processing of large number of samples under tight energy and hardware footprint constraints. Previous work has explored different hardware technologies, such as ARM and DSP embedded processors [14] and customized hardware (ASIC), to improve the performance and energy efficiency of different ECG peak detection algorithms [15]-[17]. But no previous work has considered design of reliable and energy-efficient hardware platforms that enable *continuous analysis and fusion of multiple signals on wearable devices*.

In this paper, we propose *a fault-tolerant hardware architecture for robust real-time heart rate (HR) monitoring*, using fusion of Kalman-filter-based HR estimates from ECG and ABP signals. The following novel strategies are employed to enable efficient and fault-tolerant HR monitoring with small hardware footprint and low energy consumption:

1) An optimized peak detection algorithm that can be dynamically configured for either ECG or ABP analysis and robust HR estimation based on the two signals, designed by sharing core computational kernels between two well-known peak detection algorithms [9][18], thus reducing the hardware footprint by about 38%.
2) Coarse-grained reconfigurable functional units (FUs) designed for the shared computational kernels, which can be programmed within a few cycles to perform different kinds of biomedical signal processing and enable energy-efficient computations in real time. The FUs are designed by following a template with the same interface to allow architecture extension to support other biomedical applications, such as breathing rate monitoring.
3) A low-overhead hardware fault detection and recovery unit (FDRU) that monitors the activities of FUs using a configurable watchdog timer and patient-specific invariant checking [19], and upon detection of faults will reset and re-execute the affected FU in real time to dynamically recover from the unexpected faults.

We implemented the proposed hardware architecture running the heart estimation algorithm both on an FPGA platform and as an ASIC design, and evaluated its runtime performance and energy consumption in comparison to a software implementation on an Android device. Real ABP and ECG signals from the MIMIC II database [20] were used for evaluation of the algorithm. The overheads of the fault detection and recovery mechanisms (FDRU) were evaluated by comparison to the baseline architecture. The fault coverage

of the proposed mechanisms was evaluated using simulation-based fault injection into the fault-tolerant architecture.

II. HEART RATE ESTIMATION ALGORITHM

For robust heart rate estimation with a small hardware footprint, we leveraged a technique based on the fusion of heart rates estimated from ABP and ECG signals [4][12]. More specifically, we applied algorithmic optimizations to the heart rate monitoring flow proposed in [4] to enable sharing of the computation blocks between two separate ABP and ECG processing flows to minimize the hardware area.

As shown in Fig. 1.a, there are two main stages in the monitoring flow: (1) detection of heart beats (peaks) based on the raw ABP and ECG waveforms; and (2) Kalman-filter-based estimation of heart rates and fusion of the two estimates by a weighted voter based on the signal qualities. The shared computation blocks between the ABP and ECG peak detection and heart rate estimation stages are shaded in Fig 1.a. Fig 1.b shows the new heart rate monitoring flow that is shared by the ABP and ECG signals. The heart rate is estimated over a 10-second window of each signal with a sampling rate of 125 Hz.

A. Peak Detection

For the ABP signal, a threshold-based onset detection [18], and for the ECG signal, the Pan Tompkins QRS detection algorithm [9] was used. We identified the computation steps shared between the two algorithms and modified them to create a new threshold-based peak detection algorithm that can be configured to detect peaks for both ABP and ECG signals.

The peak detection algorithms for ABP and ECG signals go through the following shared steps (shown in Fig 1.a):

1) Low-pass filtering (LPF) to remove high-frequency noise.
2) Derivative, squaring, and moving-average integration (for ECG) or slope sum calculation (for ABP) to emphasize higher-frequency peaks and enhance peak amplitudes.
3) Adaptive threshold-based peak detection to locate peaks.

Next, we will describe each of those steps in the new (optimized) peak detection algorithm (shown in Fig 1.b).

1) Low-Pass Filtering: The low-pass filters designed for ABP and ECG signals have different cut-off frequencies, as the two signals have different natural frequencies and noise interference. Using the technique proposed in [6], we designed lightweight non-recursive digital filters (FIRs) with integer multipliers and linear phase characteristics for ABP and ECG signals. The designed LPFs are described as follows:

$$\text{for ABP:} \quad y_n = (x_n + 2x_{n-1} + x_{n-2})/4$$
$$\text{for ECG:} \quad y_n = (x_n + 2x_{n-1} + 3x_{n-2} + 2x_{n-3} + x_{n-4})/9$$

Both the LPFs have the same computation structure, composed of the weighted sum of the most recent signal values with integer coefficients. For computation and design efficiency, we designed a shared LPF hardware block that can be dynamically configured by providing the coefficients and delay parameters for filtering either ABP or ECG signals.

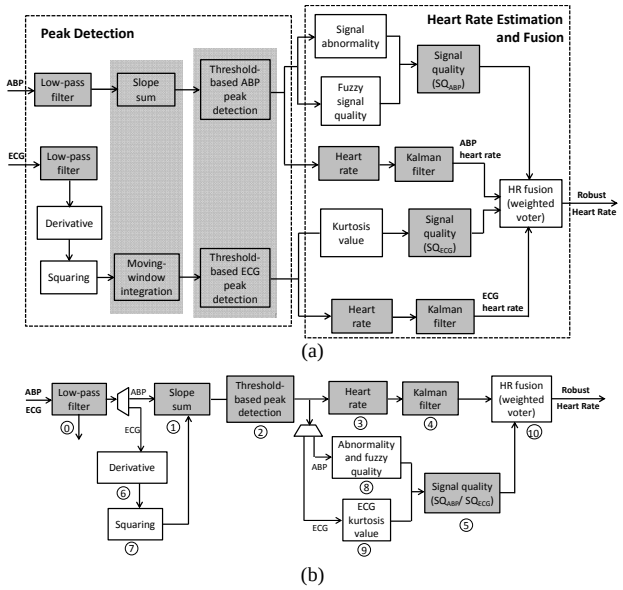


Fig. 1. Robust heart rate estimation flow

2) Peak Amplitude Enhancement: Through simulation of the Pan Tompkins QRS peak detection and ABP onset detection algorithms using ECG and ABP data collected from several patients in the MIMIC II database, we found that the moving-window integration step in the Pan Tompkins algorithm [9] can be replaced by the slope sum function used in the ABP onset detection algorithm [18] and still acheive similar or better results. Therefore, we use the slope sum as a common computation block in this step. The slope sum function is described as follows [18]:

$$SSF(k) = \sum_{i=k-w}^{k} \Delta y_i\,, \qquad \Delta y_i = \begin{cases} \Delta x_i & if\ \Delta x_i > 0 \\ 0 & if\ \Delta x_i < 0 \end{cases}$$

where k is the sample index and w is the slope sum window (w should be set as the duration of the signal's rising portion.) We chose w=15 (120 ms) for ABP and w=10 (80 ms) for ECG, because ECG has sharper peaks and rises faster than ABP.

3) Adaptive Threshold-based Peak Detection: Our peak detection method consists of three steps:

a) The onset of the slope sum signal is detected by checking a threshold (Th_{onset}) on each data sample. Then a local search is performed around the detected onset to find the local maximum (slope sum peak) and minimum values. The searching radius is half of the estimated peak to peak interval (T_{est}). Then, the difference between the slope sum's local max and min values is calculated. If the difference exceeds another threshold (Th_{diff}), the slope sum peak is accepted [18].

b) A backward search on the original signal around the slope sum's peak location is performed to detect peaks in the original signal. The backward search radius is $0.25T_{est}$.

c) To dynamically adapt to signal changes, the following weighted sum function is used to update parameters Th_{onset}, Th_{diff}, and T_{est} with the newly detected peak values:

$$V = 0.875 * V + 0.125 * V_{new\_detect}$$

The parameters are patient-specific and learnt during a training period (the first 20 windows of the data). Th_{onset} is initialized to be twice the mean slope sum value in the training period. T_{est} is the mean peak to peak intervals. Th_{diff} is half of the mean max and min slope sum difference of each window.

Table I shows the results of using the new algorithm versus the Pan Tompkins algorithm on 10 hours of ECG data from five patients in the MIMIC II database. MD rate shows the percentage of the peaks detected by the Pan Tompkins algorithm that were missed by our algorithm, and FP rate is the percentage of the peaks detected by our algorithm that were missed by Pan Tompkins. The mismatch rates were less than 1%, which indicates the accuracy of our proposed algorithm.

TABLE I. ECG peak detection results using the proposed algorithm

Patient No.	MD Rate (%)	FP Rate (%)
a40076	0.27	0.12
a41287	1.11	0.61
a41770	0.05	0.01
a42022	0.40	0.21
a42157	0.05	0.04

Pan-Tompkins Matlab code: http://www.mit.edu/~gari/teaching/6.222j/DATA/rpeakdetect.m.

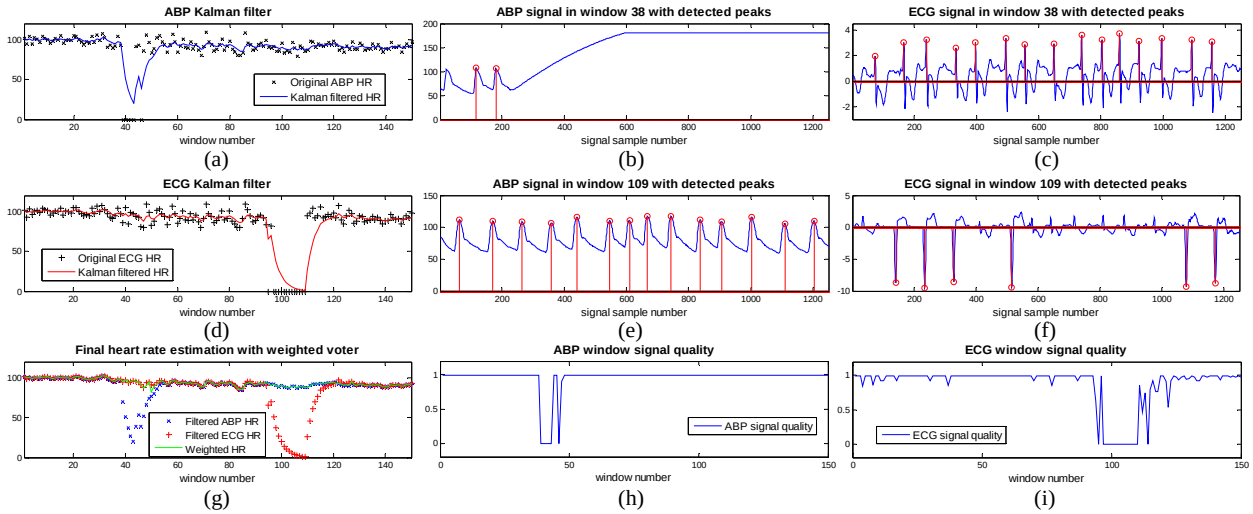


Fig. 2. Kalman-filtered heart rates and weighted heart rates for 150 windows based on ABP and ECG signals and their signal qualities

B. Heart Rate Estimation and Fusion

For heart rate estimation and fusion based on the peaks detected in the two signals, the following steps are performed:

1) The average heart rate in the current window is determined by calculating the average peak-to-peak intervals based on both the ABP and ECG signals.
2) The ABP and ECG signal qualities are estimated using the signal abnormality index, fuzzy-logic-based signal quality metric, and kurtosis, as proposed in [7], [11], and [21].
3) The Kalman filter is applied to remove high-frequency noise in the heart rate estimations between windows [4].
4) The heart rate estimates from the ABP and ECG signals are fused by a weighted voter to obtain the robust heart rate estimate for the current window. The weight for each estimate is defined based on the parameters described in [4]: (1) quality of the signal, (2) Kalman residuals for the heart rate estimates, and (3) inverse quality of the other signal. So the final estimate would always rely more on the heart rate estimated from the higher-quality signal.

Fig. 2 illustrates an example in which heart rate was estimated for 150 windows of ABP and ECG data from patient a41709 in the MIMIC II database. In this example, we see that the Kalman filter is able to remove high-frequency noise in the heart rate estimations (Fig 2.a and 2.d). At around window 40, the ABP signal is corrupted by a large artifact due to sensor disconnection, so no peaks are detected and the signal quality is very low. Similarly, around window 100, the ECG signal is noisy, and QRS peaks cannot be accurately detected. The peak detections in the ABP and ECG signals of an example window in the two segments are shown in Fig 2.b and 2.c, and Fig 2.e and 2.f, respectively. The corresponding ABP and ECG signal qualities in the 150 windows are shown in Fig 2.h and 2.i. As a result, the weighted voter masks the interference of artifacts and noise in the signals by weighting less on the low-quality signals at those segments (see Fig 2.g), so an accurate continuous estimation of heart rate is achieved.

III. RECONFIGURABLE HARDWARE ARCHITECTURE

The proposed hardware architecture runs the robust heart rate estimation algorithm discussed in Section II. It consists of three main parts: (1) an *ASIC accelerator* composed of a set of configurable functional units and a fault detection and recovery unit (FDRU), (2) a lightweight *MIPS controller*, and (3) a *shared on-chip memory system* (see Fig 3). The inputs are the raw ABP and ECG signals collected from the biomedical sensors, which are stored in the dedicated memory locations (Fig 3.a). The output is the weighted heart rate estimated from the analysis of ABP and ECG signals.

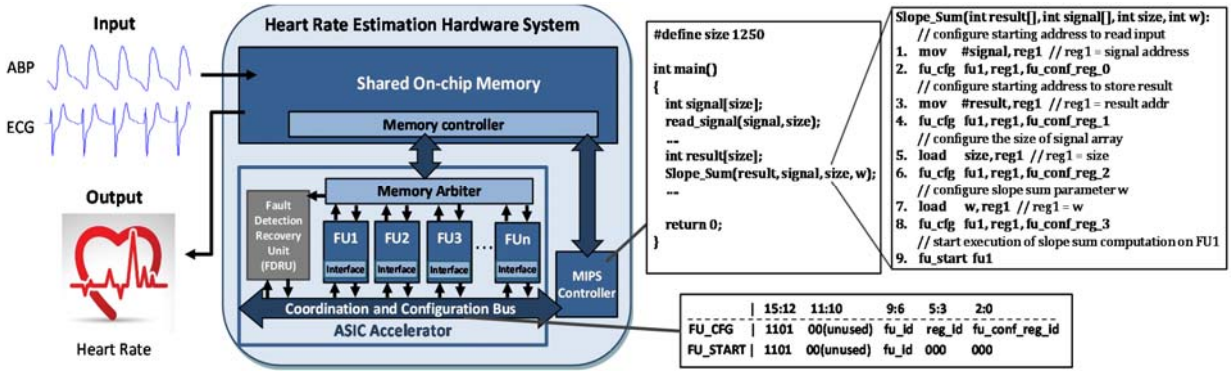
A. Functional Unit Design and Configuration

Each common computational kernel shared between the ABP and ECG flows (Fig 1.b) is implemented as a functional unit inside the architecture. Functional units (FUs) are a set of coarse-grained reconfigurable accelerators that provide efficient ABP and ECG signal processing. With algorithmic optimizations presented in Section II, a total of 11 FUs are needed for our heart rate estimation algorithm (FU numbers are highlighted next to the computational blocks in Fig 1.b). All FUs are designed according to a design template and have the same interface. More specifically, each FU is composed of three parts: (1) configuration and memory interfaces, (2) configuration registers (CRs), and (3) computation logics (a data path and a state machine controller). The FU design template enables extension of the architecture with other FUs to potentially support other monitoring applications, e.g., breathing rate estimation by fusion of photoplethysmographic (PPG) and ECG waveforms or fusion of pressure signals [22].

The FU configuration interface connects to the system coordination and configuration bus and monitors the execution and configuration instructions sent from the MIPS controller. Upon receiving a configuration instruction, the configuration interface reads the instruction from the bus, parses the configuration parameters, and configures the configuration registers. When the execution instruction is received, the configuration interface notifies the FU computation logic to start execution. The FU memory interface is responsible for reading and writing data from and to the on-chip memory shared between the MIPS controller and the ASIC accelerator.

FU configuration registers (CRs) are used to pass the needed input parameters into the FU computation logics. The most common FU parameters are the memory address to read input data, the memory address to store computation result(s), and the size of the input data to process. In addition, more configuration registers can be added in an FU to pass other useful parameters, such as the threshold values (Th_{onset}, Th_{diff}, T_{est}) for the peak detection algorithm.

FU computation logic consists of a computation data path and a state machine controller. The data path is FU-specific and implements the corresponding FU computation logic. The



(a) (b) (c)

Fig. 3. (a) Input and output of the proposed hardware; (b) hardware system overview; (c) code example and extended instructions

state machine is a hard-wired controller that schedules FU computation and memory requests through the memory interface. The state machine controllers are similar in all FUs, as the FUs follow the same scheduling pattern. FU computations are pipelined and optimized in each of our FU implementations.

B. MIPS Controller

The MIPS controller is a lightweight processor (only about 3.5% hardware area) with 16-bit instruction and 32-bit data lengths. It is responsible for (1) configuring the FUs by sending configuration parameters, (2) scheduling the FUs' execution by sending instructions, and (3) executing basic MIPS instructions that are needed between FU executions for control flow.

The FU configuration and execution instructions sent from the MIPS controller are realized by extending the MIPS basic instruction set. Two new instructions are added to the base instruction set, as shown in Fig 3.c (lower part): (1) *FU configuration* (FU_CFG), which moves a configuration parameter from a MIPS register (reg_id) to a configuration register (fu_conf_reg_id) of an FU (fu_id), and (2) *FU execution* (FU_START), which notifies an FU (fu_id) to start execution. Once an FU finishes the execution and stores the result(s) to the shared memory, it notifies the MIPS controller by sending a "DONE" signal.

Fig 3.c (upper part) shows an example of C code running on the MIPS controller. An extended version of the MIPS C compiler can be used to generate the assembly code from C programs. The FU computations are invoked in the C program as intrinsic functions that are recognizable by the compiler. The compiler needs to maintain a table of mappings between C intrinsic functions and the corresponding FUs and know the meaning of FU configuration registers. In the given example, the *Slope_Sum* intrinsic function is called in the C program. The compiler maps it to FU1, which is responsible for slope sum computation, and generates the MIPS assembly code shown on the right. In lines 1-8, FU1 is configured by FU_CFG instructions, and in line 9, the execution is started. During execution, FU1 reads the signal sample values from the memory locations specified by CR0, computes slope sum values with the configured slope sum window (CR3), and writes the results to the result array (CR1). The total signal size to be read and processed is specified by CR2.

IV. FAULT TOLERANCE MECHANISMS

A low-overhead hardware fault detection and recovery unit (FDRU) has been designed to protect the functional units against unexpected transient faults (Fig. 4). Since the FUs account for more than 93% of the area and 92% of the energy consumption in the processing part of the hardware architecture (ASIC Accelerator + MIPS controller), the FDRU is able to cover most of the hardware.

The fault model we simulated for evaluation of the fault-tolerance mechanisms is the low-level transistor fault that may flip the result of a logic gate and then propagate to affect the application's output. Only transient faults are considered, because transient faults or soft errors are the most common hardware faults, the rate of which is expected to increase 8% per logic state bit in each transistor generation [23].

Upon detection of a fault, the corresponding FU is reset and re-executed. Therefore, with the proposed detection and recovery mechanism, both transient and permanent faults can be detected, but only transient faults can be recovered from.

A. Fault Detection

The FDRU uses two kinds of detectors, the configurable watchdog timer and patient-specific invariant checking [19], to detect hangs in the control logic and faults in the computation logic. The watchdog timer monitors the execution time of each FU and times out on FU hangs. Invariants are the conditions that hold true during the FU's execution, and if any invariant is violated, it means a fault has happened. Two kinds of invariants are used for our proposed fault detection: *result invariants* and *address invariants*. Therefore, the FDRU is able to detect faults that cause the FU:

1) to hang (not finish the execution within the amount of time specified by the watchdog timers);
2) to generate incorrect results that violate the FU's *result invariants*; or
3) to write results to incorrect memory addresses that violate the FU's *address invariants*;

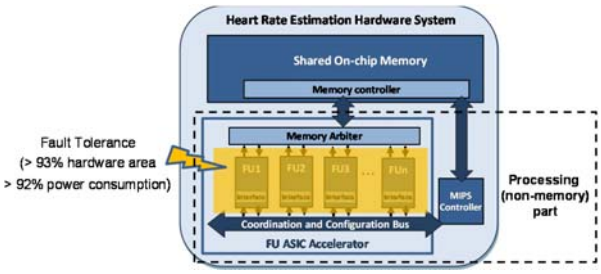


Fig. 4. Fault tolerance hardware coverage

The ***watchdog timer*** is a module inside the FDRU that can be dynamically reconfigured upon execution of each FU to detect hangs due to faults in the control flow. If the number of cycles since the start of an FU execution goes beyond the configured threshold, the FDRU assumes the FU has hung.

Address invariants are obtained during application compiling. The compiler assigns memory locations for each FU to write the results. Some FUs generate only a single result, e.g., FU3 (heart rate), FU5 (signal quality), and FU10 (weighted voter). So their results are written at fixed memory locations that can be used as their address invariants. Some other FUs generate an array of results, e.g., FU0 (low-pass filter) and FU1 (slope sum), and write the result elements to a range of memory addresses with a given stride (for example one). Therefore, the memory range and the stride between consecutive memory writing locations are used as those FUs' address invariants.

Result invariants are obtained by patient-specific application profiling. Since each FU is responsible for a processing step in the heart rate estimation algorithm, the output of each FU has a specific application-level semantic (in contrast to the output of basic instructions, such as addition, and subtraction, in a general-purpose processor). Therefore, we utilize that property to obtain FU result invariants for fault detection. Two kinds of values are profiled for result invariants: the range of results and the difference between two consecutive results generated by the FU. Table II lists an example set of result invariants for different functional units. They are obtained by profiling data from patient a40050 in the MIMIC II database. Just like the threshold parameters used in the heart rate estimation algorithm, result invariants are signal- and patient-specific. Therefore, FUs are also designed to be configurable for result invariants obtained from profiling.

It should be noted that corrupted or abnormal signal inputs may also cause violations in result invariants if the scenario was not profiled. When that happens, abnormal signal values are detected as FU faults. Since it may not be possible to profile all patient and sensor input scenarios, there is a trade-off between the fault detection coverage and the false detection rate. If the invariants are set too tight (they fit only a small set of profiled data), an FU fault may be incorrectly detected upon new data samples. For example, in the worst case, if a patient is completely healthy during the profiling phase, when a problem happens to the patient later and changes the pattern of the input data, this may result in FU fault detection instead of patient problem detection. On the other hand, if the invariants are set too loose, FU faults may not cause violations of the result invariants, and this may result in undetected faulty heart rate estimation.

Fig. 5 shows an example with three result invariants. As the profiling period increases, the range of the result invariants increases. The profiling data from 1000 to 4000 windows includes 11 occurrences of physician-annotated arrhythmia

TABLE II. FU result invariants (profiled with data from 1000 windows)

FU number		Result Invariant			
		min	max	min_{diff}	max_{diff}
FU0 – low pass	for ABP	32.400	176.100	-18.600	23.400
	for ECG	-6.315	7.250	-4.160	4.015
FU1 – slope sum	for ABP	0	87.300	-20.700	52.200
	for ECG	0	18.180	-16.621	18.180
FU2 – peak detection	peak index	0	1240	45	322
	peak number	1	17	-8	13
FU3 – heart rate		53.354	104.530	-37.190	40.396
FU4 –Kalman filter	filtered value	26.350	100.549	-25.088	17.991
	residue	-92.866	66.596	-	-
FU5 – signal quality		0	1.000	-	-
FU6 – derivative		-4.160	4.015	-6.375	8.175
FU7 – squaring		0	17.306	-11.136	16.621
FU8 – ABP beat quality		0	1.000	-	-
FU9 – ECG beat quality		2.158	8.984	-	-
FU10 – heart rate fusion		86.733	100.513	-1.495	2.583

Note: the following conditions hold true during the corresponding FU execution: $y_i \geq min,\ y_i \leq max,\ y_i - y_{i-1} \geq min_{diff},\ y_i - y_{i-1} \leq max_{diff}$, where y_i is the current result value and y_{i-1} is the previous result value.

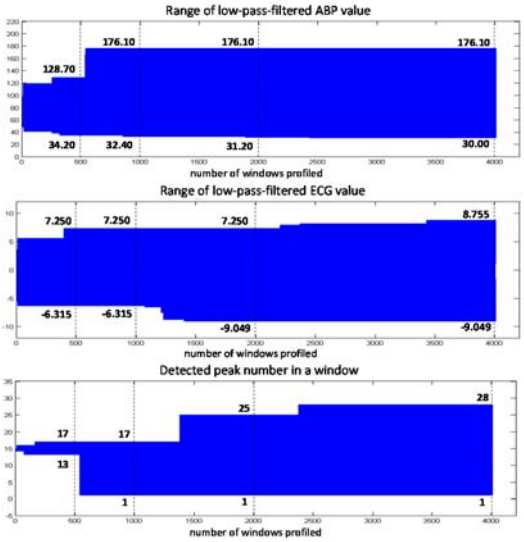


Fig. 5. Relationship between the result invariant range and the profiled data window number (with three example invariants)

alarms. Even though the invariant range becomes larger, the experimental results in Section IV.B show that the fault detection coverage is only slightly affected. The reason is that the manifested hardware faults usually change the FU result by a large amount (beyond the patient's physiological ranges).

Therefore, medical knowledge about the patient should be combined with profiling data to set result invariants for some of the FUs, such as the low-pass filter, slope sum, and peak detection. The patient's physiological ranges can be used to find the result invariants (e.g., the patient's blood pressure never goes above 200 or below 30). Some other FUs, such as the signal quality, do not need the physiological ranges to find the result invariants, because their results should always be within a certain range no matter what the inputs are (e.g., signal quality must be a numeric value between 0 and 1).

On the other hand, false detections of faults are not always harmful, because they could be a symptom of severe signal corruptions caused by sensor disconnections. For example, if the invariants have been set via profiling of a long period of data with the patient's physiological ranges considered, and a detected fault was not actually caused by a nonexistent hardware fault that was erroneously detected, it will be certain that the detected fault was caused by input that was outside the patient's physiological range and very likely resulted from severe signal corruptions that are worth the physician and patient's attention.

The FDRU is part of the ASIC accelerator (Fig. 3.b) and monitors all the FU configurations and executions. During FU configuration, the FDRU keeps a copy of all configuration register values, which will be used for fault recovery upon detection of a fault. When an FU sends a request to the memory arbiter to write a result, the FDRU checks it with both the address and result invariants of this FU using hardware range checkers.

B. Fault Recovery

During the execution of a functional unit, if a fault is detected, the FDRU takes three steps to recover the faulty FU: (1) it resets the FU by sending a reset signal to it. (2) it reconfigures the FU by sending configuration instructions (FU_CFG), and (3) it re-executes the FU by sending the execution instruction (FU_START).

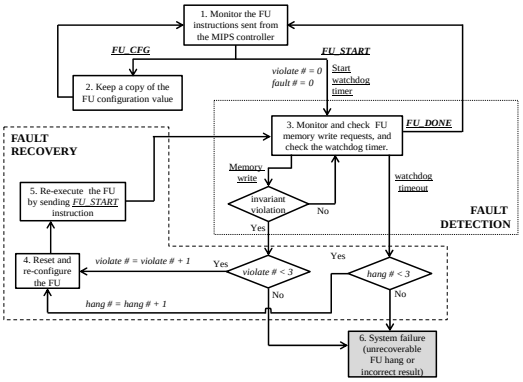


Fig. 6. Fault Detection and Recovery Unit (FDRU) operation flow

Fig. 6 illustrates the entire operation flow of the FDRU. It includes both the fault detection and recovery mechanisms discussed above. As shown in Fig. 6, if either the invariant is violated three times or the hang is detected three times during a MIPS-scheduled execution, the FDRU enters the system failure state and notifies the user about the failure. This will happen if there is a permanent hardware fault or the transient fault continually occurs in re-executions. If the system failure state was entered because of three consecutive transient faults, a reset of the whole architecture may fix the problem. However, a permanent fault can only be detected by the FDRU, but cannot be recovered from.

V. EXPERIMENTAL RESULTS

The proposed hardware architecture has been implemented both on a Xilinx FPGA platform and as an ASIC design using the Synopsys Design Compiler. ASIC is the target platform for the final product (the proposed heart rate monitor), while the FPGA is used as a platform to evaluate the proposed hardware architecture and to experiment with its extension with more FUs for other potential applications (heart rate estimation application).

For comparison with the off-the-shelf embedded processors, we also implemented the same heart rate estimation algorithm as an Android application on a Nexus 7 tablet (2013 model), equipped with the Qualcomm Krait processor (architecturally similar to ARM Cortex-A15).

Table III lists the experimental setup of the three platform implementations (Android, FPGA, and ASIC). The Android application ran on the Qualcomm Krait processor of the Snapdragon chipset at 1.5 GHz. The execution time of the Android application was recorded by inserting time measurement functions in the code immediately before and after the heart rate estimation algorithm. The Android power consumption was profiled using the Qualcomm Trepn Profiler. During the measurements of the execution time and power consumption, all the other Android applications and services were turned off.

The Xilinx Virtex 5 ML507 board (XC5VFX70T) was used as a platform for the FPGA implementation. We were able to run the proposed hardware architecture on the actual FPGA platform, while the FPGA results were collected from the simulation of the FPGA-synthesized hardware. The application's execution time on FPGA is calculated by multiplying the number of execution cycles (from the cycle-accurate Modelsim simulation) by the clock period (from the Xilinx ISE timing report). The FPGA power consumption was profiled using the Xilinx Power Analyzer based on the signal activities collected from the post-routing simulations.

TABLE III. Experiment toolsets for hardware system evaluation

Platform	Frequency	Design Tools	Evaluation Tools
Android	Snapdragon S4 @ 1.5 GHz	Android SDK (test and evaluate on the 2013 Asus Nexus 7 tablet)	Qualcomm Trepn Profiler[1]0
FPGA	66.6 MHz	Xilinx ISE 14.2 (test and evaluate on the Virtex-5 XC5VFX70T FPGA)	Modelsim SE 10.1a, Xilinx ISE, and Xilinx Power Analyzer[2]
ASIC	100 MHz (up to 222.2 MHz)	**Processing logics:** Synopsys Design Compiler with NanGate 45 nm Open Cell Library. **On-chip memory:** Synopsys Generic Memory Compiler (32 nm).	Modelsim SE 10.1a and Synopsys Design Compiler with NanGate 45 nm and Generic Memory Compiler 32 nm technology libraries[3]

1. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler.
2. http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools/v2012_2---14_2.html.
3. http://www.nangate.com/?page_id=2325

For the ASIC implementation, separate tools were used for synthesizing the processing logic and on-chip memory (Fig. 4). Just as in the FPGA implementation, the ASIC results were collected from the simulation of the synthesized hardware. A 100 MHz clock frequency was used for the ASIC implementation. The power consumption of ASIC was profiled using the tools in the Design Compiler based on the signal activities collected from the post-synthesis simulation.

We first evaluated the performance and energy consumption of the baseline hardware architecture (without the FDRU) in comparison to Android, FPGA, and ASIC implementations. Then we evaluated the proposed fault tolerance mechanisms by measuring the overhead and detection coverage of the FDRU.

A. *Baseline Hardware System Evaluation*

The resource utilizations for the FPGA and ASIC implementations are shown in Table IV. The computational steps shared between ECG and ABP monitoring flows utilized up to 38% of the hardware resources, which were saved in the proposed heart rate monitoring flow (Fig. 1.b).

Fig. 7 shows a comparison of the runtime performance and energy consumption of the three platforms, obtained by running 1000 windows (10,000 seconds) of patient data from the MIMIC II database. All the results are normalized to the results of the ASIC platform, which are listed in Table V.

Compared to ASIC, the execution times to process the same amount of patient data on the Android and FPGA platforms are 20.62 and 1.50 times longer, respectively. The speedup of ASIC compared to the Android implementation is mainly from (1) the efficiency of the FU modules that are optimized in the ASIC logic, and (2) the faster memory accesses enabled by the on-chip memory. On the other hand, since the same underlying hardware design is used for both the FPGA and ASIC implementations, the hardware cycles to run the same application are the same on both platforms. So the speedup of ASIC compared to FPGA is due only to the higher clock frequency.

Energy consumption directly affects battery life. Both the ASIC and FPGA implementations of the proposed hardware system are much more energy-efficient than the Android implementation. They consume 1/2871 and 1/923 of the energy used by Android, respectively. The reason is that the Android implementation runs on the general-purpose embedded processor (Krait) with complicated processing pipelines and hierarchical memory systems, which are designed to reduce the processing latency of general embedded applications.

TABLE IV. Resource utilizations of the proposed hardware system

Processing logics	FPGA	ASIC
Total	11,856 LUTs[a] + 22 DSP48E	53,697 cell gates (0.121 mm^2)
Shared logic (FU0–FU5) (%)	2,718 LUTs[a] + 4 DSP48E (38.1%)	16,844 cell gates (31.4%)
On-chip memory	32 KB Block RAM	32 KB SRAM (0.195 mm^2)[b]

[a] LUT means "look-up table" (LUT is 6-input for the Virtex 5 FPGA family).
[b] The ASIC on-chip memory is implemented with the 32 nm SRAM generated by the Generic Memory Compiler (a different technology library from the one used for processing logics).

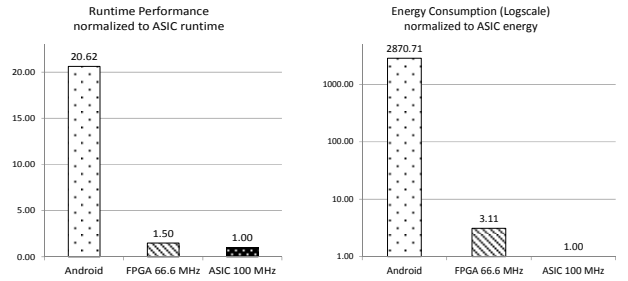


Fig. 7. Comparison of runtime performance and energy consumption (for each platform, the same heart rate estimation application was run for 1000 windows of patient data (a40050) from the MIMIC II database.)

TABLE V. Runtime and energy consumption of the ASIC implementation

Runtime (s)	Energy (mJ)	
0.211	**Processing logic**	1.453
	On-chip memory	0.096
	Total	1.549

On the other hand, the proposed hardware architecture is composed of FUs that have been specially designed and optimized for the target monitoring application. In addition, because of the small memory size required, the on-chip (cache-like) memory is directly used as the main memory in the proposed hardware, which simplifies the memory system design and reduces the corresponding energy consumption. Both the ASIC and FPGA implementations benefit from those application-specific optimizations. Therefore, with the same battery capacity, the proposed hardware architecture on the ASIC and FPGA platforms would be able to achieve, respectively, 2871 and 923 times the battery life of an implementation on the Android platform.

B. Fault Tolerance Evaluation

The overheads of the FDRU in resource utilization, power consumption, and runtime performance are listed in Table VI. During a fault-free execution, the fault-tolerant architecture consumes 37.01% and 33.89% more power, compared to the baseline architecture, in the FPGA and ASIC implementations, respectively. The extra power consumption is mainly due to the invariant checking and watchdog timers used for fault detection.

The FDRU does not incur performance overheads (any extra hardware cycles) during normal monitoring without faults, because the fault detection checking is executed in parallel with normal FU executions, and none of the fault detection checking is on the critical execution path. When faults are detected during an execution, extra power and performance overheads would be introduced by the fault recovery process. The amount of extra overhead depends on the frequency of the FU fault detection and the specific FU to which the fault occurs.

The fault-tolerant architecture utilizes about 14.5% more look-up tables in FPGA and 15.5% more cell gates in ASIC, compared to the baseline architecture. The resource overhead is due to (1) the FDRU's controlling state machine logics (Fig. 6), (2) a copy of all FU configuration registers, (3) the watchdog timer, and (4) the FU address and result invariants, as well as the invariant checkers (hardware comparators).

TABLE VI. Overheads of FDRU in FPGA and ASIC implementations

	FPGA	ASIC
Power*	37.01%	33.89%
Performance*	0%	
Resource	14.54% LUTs	15.54 % cell gates (12.65% area)

* Power and performance overheads in the table are the overheads during normal monitoring when no fault has been detected. If there are detected faults, more overheads would be introduced due to FU re-configuration and re-execution in the fault recovery process of FDRU.

TABLE VII. Description of possible fault injection results

	Baseline Architecture (without FDRU)	Fault-Tolerant Architecture (with FDRU)
Correct Result	The hardware finishes execution on time*, and the heart rate detected is correct (faults are not manifested).	
Incorrect Result	The hardware finishes execution in time*, but the heart rate detected is incorrect (faults are manifested).	
System Failure	The hardware does not finish execution on time*.	Either the FU hang or the invariant violation is detected three times in a single MIPS-scheduled execution of the FU.

* "on time" means within 5 times the supposed execution time, which is the execution time of the same application without fault injection.

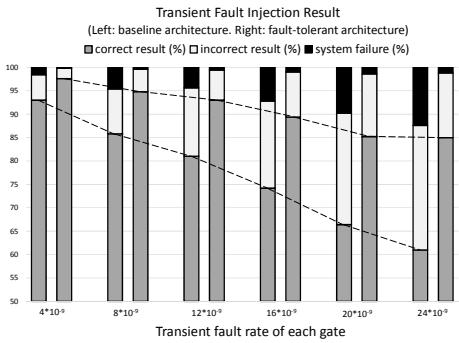


Fig. 8. Fault injection result comparison between the baseline system and the fault-tolerant system (with the proposed FDRU)

To evaluate the detection coverage of the proposed fault tolerance mechanisms, we used the CrashTest fault injection framework [24] to inject transient faults to all synthesized logic gates of the FUs (highlighted in Fig. 4). We injected faults at different fault rates of 4×10^{-9} to 24×10^{-9} per logic gate per cycle. For each fault rate, 500 simulations were performed, with faults randomly triggered at different gates in each simulation. Depending on the time and location of which the fault was triggered, we observed three possible results, listed in Table VII.

Fig. 8 shows a comparison of the fault injection results on the baseline architecture versus the fault-tolerant architecture with the proposed FDRU. The result invariants were profiled with 1000 windows of patient data. The results are in terms of the percentages of the three possible results (shown in Table VII) out of the 500 simulations for each fault rate.

At all fault injection rates, the FDRU was able to increase the correct result percentages by detecting the FU faults and dynamically recovering from them through FU re-executions. In total, incorrect system behavior (incorrect results and system failures) are reduced by 55.9-65.7% in all fault injection rates (e.g., reduced from 33.6% to 14.8% under the fault rate of $20*10^{-9}$ per logic gate per cycle). In addition, all system failures were detected in the fault-tolerant architecture. If the FDRU fails, the baseline architecture will still operate normally, but with no fault protection.

As discussed in Section IV, with longer profiling period (more input scenarios), the ranges of FU result invariants become larger. As a result, the fault detection coverage would be affected, because the probability would be higher that the

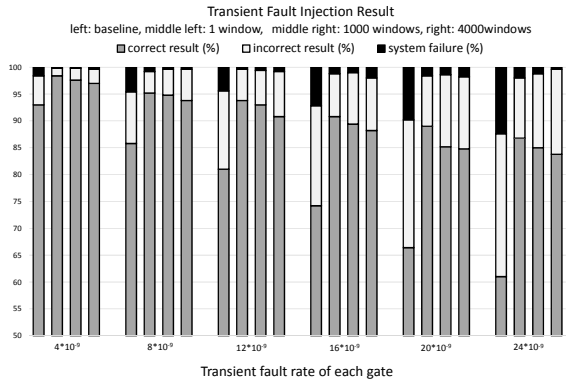


Fig. 9. Fault injection result comparison between the baseline system and the fault-tolerant system with different profiling data sizes

fault would cause the incorrect result to be within the invariant range and thus undetected. To illustrate the effect of the profiling data size on the fault detection coverage, Fig. 9 compares the fault injection results for the baseline architecture and fault-tolerant architecture with three different profiling data sizes. The comparison results show that the fault detection coverage does not decrease much as the profiling data size (result invariant range) becomes larger. Even with 4000 windows of profiling data, the incorrect system behaviors (incorrect results and system failures) are still reduced by at least 51.58% compared to the baseline system. The reason is that the detected fault usually causes the FU result to be changed by a large amount that is beyond normal physiological ranges. Therefore, the result invariants should be set based on both the patient's physiological ranges and the profiling of long periods of patient data, so that the false fault detections that are not indications of severe signal corruptions can be eliminated, while the fault detection and recovery coverage is still kept high.

VI. CONCLUSIONS

In this paper, a fault-tolerant hardware architecture for robust real-time heart rate monitoring is introduced. A signal fusion algorithm for robust heart rate estimation based on analysis of ECG and ABP waveforms is used. We developed an optimized peak detection algorithm that can be dynamically configured for detecting heart beats from either ECG or ABP signals, which enables sharing of the computational blocks and reduces hardware footprint by 38%. A fault detection and recovery unit (FDRU) is proposed that by utilizing the watchdog and patient-specific invariant checkers can protect FUs from transient hardware faults. The proposed hardware architecture is implemented on both an FPGA platform and as an ASIC device. Both implementations achieved better runtime performance (almost 20 times faster) that the same algorithm implemented on an Android device, while consuming much lower energy (1/2871 and 1/923 of Android implementation). In addition, the proposed fault-tolerant mechanisms can protect the device against 55.9%-65.7% of incorrect results and system failures, with low energy (34%), area (15%), and no performance (0%) overheads.

The proposed hardware architecture can be used as a configurable platform for robust real-time estimation of a variety of cardiovascular parameters on a wearable device. Future work will focus on evaluation of the proposed architecture for monitoring cardiac arrhythmias in real patient settings and on the comparison of detection results with the state-of-the-art heart rate monitoring algorithms and systems.

ACKNOWLEDGMENT

This work is supported in part by the Department of Energy under Award Number DE-OE0000097, by the Air Force Research Laboratory and the Air Force Office of Scientific Research under Agreement No. FA8750-11-2-0084, and by National Science Foundation under Grant No. CNS 13-37732. The authors would like to acknowledge the contributions of Yangyang Yu in hardware development and ASIC synthesis of functional units in the proposed architecture. We also thank Jenny Applequist for her assistance in preparing the paper.

REFERENCES

[1] Heartcheck ECG monitor website. Available: http://www.theheartcheck.com/.

[2] H. Alemzadeh et al., "Towards resiliency in embedded medical monitoring devices," in *Dependable Systems and Networks Workshops (DSN-W)*, 2012 IEEE/IFIP 42nd International Conference on. IEEE, 2012, pp. 1-6.

[3] M. K. Chung, et al., "Aggregate national experience with the wearable cardioverter-defibrillator event rates, compliance, and survival," *Journal of the American College of Cardiology*, vol. 56, no. 3, pp. 194-203, 2010.

[4] Q. Li, R. G. Mark, and G. D. Clifford, "Robust heart rate estimation from multiple asynchronous noisy sources using signal quality indices and a Kalman filter," *Physiol. Meas.*, vol. 29, no. 1, p. 15, Jan. 2008.

[5] H. Alemzadeh, et al., "Analysis of Safety-Critical Computer Failures in Medical Devices," *IEEE Security and Privacy, vol. 11, no. 4, pp. 14-26, July/August* 2013.

[6] P. A. Lynn, "Online digital filters for biological signals: Some fast designs for a small computer," *Med. Biol. Eng. Comput.*, vol. 15, no. 5, pp. 534–540, Sept. 1977.

[7] J. X. Sun, A. T. Reisner, and R. G. Mark, "A signal abnormality index for arterial blood pressure waveforms," in *Computers in Cardiology*, 2006, pp. 13–16.

[8] T. Berset, D. Geng, and I. Romero, "An optimized DSP implementation of adaptive filltering and ICA for motion artifact reduction in ambulatory ECG monitoring," in *Proc. Conf. IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 6496–6499, 2012.

[9] J. Pan and W. J. Tompkins, "A real-time QRS detection algorithm," *IEEE Trans. Biomed. Eng.*, vol. 32, no. 3, pp. 230–236, Mar. 1985.

[10] A. Pachauri and M. Bhuyan, "ABP peak detection using energy analysis technique," in *Multimedia, Signal Processing and Communication Technologies (IMPACT)*, 2011 International Conference on. IEEE, 2011, pp. 36-39.

[11] D. W. Zong, G. B. Moody, and R. G. Mark, "Reduction of false arterial blood pressure alarms using signal quality assessement and relationships between the electrocardiogram and arterial blood pressure," *Med. Biol. Eng. Comput.*, vol. 42, no. 5, pp. 698–706, Sept. 2004.

[12] M. Ebrahim, J. Feldman, I. Bar-Kana, "A robust sensor fusion method for heart rate estimation," *Journal of Clinical Monitoring*, vol. 13, no. 6, pp. 385–393, 1997.

[13] A. Aboukhalil et al., "Reducing false alarm rates for critical arrhythmias using the arterial blood pressure waveform," *Journal of Biomedical Informatics*, vol. 41, no. 3, pp. 442–451, 2008.

[14] I. Al Khatib et al., "A multiprocessor system-on-chip for real-time biomedical monitoring and analysis: Architectural design space exploration," in *Proceedings of the 43rd Annual Design Automation Conference*. ACM, 2006, pp. 125-130.

[15] C. Pavlatos et al., "Hardware implementation of Pan & Tompkins QRS detection algorithm," in *Proceedings of the EMBEC05 Conference*, 2005.

[16] R. Stojanovic et al., "A FPGA system for QRS complex detection based on integer wavelet transform," *Measurement Science Review*, vol. 11, no. 4, pp. 131-138, 2011.

[17] H. Alemzadeh et. al, "An embedded reconfigurable architecture for patient-specific multi-parameter medical monitoring," *Proc. Conf. IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 1896–1900, 2011.

[18] W. Zong et al., "An open-source algorithm to detect onset of arterial blood pressure pulses," *Computers in Cardiology*, IEEE, pp. 259-262, 2003.

[19] M. D. Ernst et al., "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng*, vol. 27, no. 2, pp. 99-123, 2001.

[20] G. B. Moody and R. G. Mark, "A database to support development and evaluation of intelligent intensive care monitoring," *Computers in Cardiology*, pp. 657–660, 1996.

[21] T. He, G. Clifford, and L. Tarassenko. "Application of independent component analysis in removing artefacts from the electrocardiogram." *Neural Computing & Applications*, vol. 15, no. 2, pp. 105–116, 2006.

[22] L. Tarassenko, L. Mason, and N. Townsend. "Multi-sensor fusion for robust computation of breathing rate," *Electronics Letters*, vol. 38, no. 22, pp. 1314-1316, 2002.

[23] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10-16, 2005.

[24] A. Pellegrini et al., "CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework," in *IEEE International Conference on Computer Design*, pp. 363–370, 2008.