ImplantBench: Characterizing and Projecting Representative Benchmarks for Emerging Bioimplantable Computing

HEALTHCARE WILL ADVANCE DRAMATICALLY WHEN MICRO- AND NANOSCALE COMPUTING CHIPS, IMPLANTED IN THE HUMAN BODY, CAN ASSIST DIGITALLY IN CLINICAL DIAGNOSIS AND THERAPY. TO DESIGN AND ENGINEER THE NECESSARY PROCESSOR AND ACCELERATOR ARCHITECTURES, COMPUTER ARCHITECTS MUST FIRST UNDERSTAND THE CURRENT AND POTENTIAL WORKLOADS. IMPLANTBENCH, THE FIRST ATTEMPT AT A REPRESENTATIVE WORKLOAD TAXONOMY, INCLUDES REALISTIC, FULL-BLOWN WORKLOADS SPANNING SECURITY, RELIABILITY, BIOINFORMATICS, GENOMICS, PHYSIOLOGY, AND HEART ACTIVITY.

Zhanpeng Jin
Allen C. Cheng
University of Pittsburgh

•••••• The idea of implanting micro- and nanoscale electronic devices in the human body to collect, process, and communicate biological data for clinical and augmentative applications provides an exciting example of how the confluence of information technology, biotechnology, and nanotechnology can revolutionize healthcare in the 21st century. For example, a retina implant could restore vision to people with visual impairment (see http://artificialretina.energy.gov). Patients with spinal cord injuries who would otherwise be permanently paralyzed could regain the ability to move their limbs through functional electrical stimulation (FES) implants and other neural prosthetics. Individuals with memory formation deficits could recover the ability to form new memories using an artificial hippocampus. Implanted radio-frequency identification chips (RFIDs) could let emergency room caregivers retrieve physiological characteristics and perform health diagnosis for unconscious arrivals to speed their treatment and perhaps save their lives.

The deployment status of these bioimplantable devices is at various stages. Some, such as deep brain stimulation (DBS), FES, cochlear, and RFID implants, have been clinically adopted or approved by the US Food and Drug Administration (FDA); others, such as retina implants and the

Related Work on Benchmarks

Computer architects have long used benchmark programs representative of real programs to assess the performance characteristics and power consumption of their processor designs. Generally speaking, the currently existing benchmark suites fall into three categories: general-purpose computing benchmarks, embedded-system benchmarks, and other application-specific benchmarks.[1] Here, we briefly review some of the most significant and widely adopted benchmark suites in industry and academia.

There is no doubt that benchmark suites released by the Standard Performance Evaluation Corporation (SPEC) are the most popular and widely used commercial evaluation tools. To date, the SPEC CPU-series benchmark suite has been the incontestable source for benchmarking general-purpose computing systems.[2] However, although it has gone through several generations—from SPEC CPU1989 to SPEC CPU2006 and beyond—SPEC still can't be applied to all application-specific domains without introducing redundant or irrelevant programs. We draw this conclusion from the study conducted by Vandierendonck and De Bosschere, who used principal component analysis (PCA) to deduct that even the well-established SPEC CPU2000 suite included some eccentric benchmarks; that is, it is possible to speed up the benchmarks significantly by means of an otherwise irrelevant optimization, because the execution time of some benchmarks is determined to a large extent by one specific bottleneck.[3]

Owing to this situation, a mass of application-specific benchmark suites have emerged in the last decade. Among them, MiBench, proposed by researchers from the University of Michigan, establishes one of the most representative evaluation criteria for benchmarking embedded systems.[4] Following the model established by the *EDN* Embedded Microprocessor Benchmark Consortium (EEMBC), the MiBench benchmarks are divided into six categories of application domains—automotive and industrial control, consumer devices, office automation, networking, security, and telecommunications. Other embedded application benchmark suites include

- EEMBC, MiBench's predecessor, which covers automotive, consumer and digital entertainment, networking, office automation, and telecommunication applications;[5]
- CommBench, NetBench, and NpBench for network applications;[6–8] and
- MediaBench, proposed by researchers at UCLA, for multimedia applications.[9]

In addition to these, various other benchmark suites have been developed to evaluate specific applications. Nazhandali, Minuth, and Austin developed the SenseBench suite to facilitate evaluation of wireless sensor network processors.[10] For simulation-based computer architecture research, KleinOsowski and Lijia proposed the MinneSPEC benchmark workload,[11] which they derived from the standard SPEC CPU2000 workload. Other benchmark suites include 3DMark for 3D applications,[12] PacketBench for implementing network processing applications and obtaining an extensive set of workload characteristics,[13] and PennBench for embedded Java programs.[14]

For the emerging interdisciplinary field of bioinformatics, two separate research groups are proposing benchmark suites: Albayraktaroglu et al. have proposed BioBench;[15] and Bader, Li, et al. have developed BioPerf.[16] Both benchmark suits are composed of bioinformatics workloads such as DNA/RNA sequence comparison, phylogenetic reconstruction, protein structure prediction, and sequence homology.

Thus, although numerous benchmark suites have been developed and proposed to meet the different needs of specific application domains, to the best of our knowledge there is no prior effort that attempts to construct a comprehensive, representative workload taxonomy for emergent bioimplantable computer systems. Without this workload taxonomy, we as a community do not have the necessary exploration vehicle with which to assess the requirements for future bioimplantable systems. This is what led us to develop ImplantBench.

artificial hippocampus, are still in research and clinical-trial phases. Despite their varied deployment status, these bioimplantable devices share one thing: very limited computing power.

As was the case with many electronic devices in their early stages (televisions, phones, cameras, audio and video players and recorders, and even computers), the computational ability of the current bioimplantable device is very crude. This computational crudeness falls broadly into two categories:

- The overall computational tasks that the implant device carries out are underrealized.
- The actual implantable parts of the device have very limited computing power, if any.

Some examples of the first category of computational crudeness are RFID implants, whose main ''computing'' ability is to store a numeric identifier, usually 128 bits or less, and to transmit it upon request; DBS and FES implants, which generate and regulate electrical pulses for stimulation; and cochlear and retina implants, which handle the basic signal processing of auditory and optical input. Compared to the computing power (and the concomitant functional sophistication) that most modern processors could offer, bioimplantable de-

References

1. J.J. Yi and D.J. Lijia, ''Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations,'' *IEEE Trans. Computers*, vol. 55, no. 3, Mar. 2006, pp. 268-280.
2. J. Henning, ''SPEC CPU 2000: Measuring CPU Performance in the New Millennium,'' *Computer*, vol. 33, no. 7, July 2000, pp. 28-35.
3. H. Vandierendonck and K. De Bosschere, ''Eccentric and Fragile Benchmarks,'' *Proc. Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 04), IEEE CS Press, 2004, pp. 2-11.
4. M.R. Guthaus et al., ''MiBench, A Free, Commercially Representative Embedded Benchmark Suite,'' *Proc. IEEE Int'l Workshop Workload Characterization* (WWC 01), IEEE CS Press, 2001, pp. 3-14.
5. A. Weiss, ''The Standardization of Embedded Benchmarking: Pitfalls and Opportunities,'' *Proc. Int'l Conf. Computer Design* (ICCD 99), IEEE CS Press, 1999, pp. 492-498.
6. T. Wolf and M. Franklin, ''Commbench—A Telecommunications Benchmark for Network Processors,'' *Proc. Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 00), IEEE CS Press, 2000, pp. 154-162.
7. G. Memik, W. Mangione-Smith, and W. Hu, ''NetBench: A Benchmarking Suite for Network Processors,'' *Proc. Int'l Conf. Computer-Aided Design* (ICCAD 01), IEEE CS Press, 2001, pp. 39-42.
8. B. Lee and L. Kurian John, ''NpBench: A Benchmark Suite for Control Plane and Data Plane Applications for Network Processors,'' *Proc. Int'l Conf. Computer Design* (ICCD 03), IEEE CS Press, 2003, pp. 226-233.
9. C. Lee, M. Potkonjak, and W.H. Mangione-Smith, ''MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems,'' *Proc. 30th Ann. Int'l Symp. Microarchitecture* (MICRO 97), IEEE CS Press, 1997, pp. 330-335.
10. L. Nazhandali, M. Minuth, and T. Austin, ''SenseBench: Toward an Accurate Evaluation of Sensor Network Processors,'' *Proc. Int'l Symp. Workload Characterization* (WWC 05), IEEE CS Press, 2005, pp. 197-203.
11. A.J. KleinOsowski and D.J. Lijia, ''MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research,'' *IEEE Computer Architecture Letters*, vol. 1, no. 1, Jan. 2002, pp. 7-10.
12. *3DMark05 Whitepaper*, Futuremark, 2006; http://www.futuremark.com/products/3dmark05.
13. R. Ramaswamy and T. Wolf, ''PacketBench: A Tool for Workload Characterization of Network Processing,'' *Proc. Int'l Workshop Workload Characterization* (WWC 03), IEEE CS Press, 2003, pp. 42-50.
14. G. Chen et al., ''PennBench: A Benchmark Suite for Embedded Java,'' *Proc. Int'l Workshop Workload Characterization* (WWC 02), IEEE CS Press, 2002, pp. 71-80.
15. K. Albayraktaroglu et al., ''BioBench: A Benchmark Suite of Bioinformatics Applications,'' *Proc. Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 05), IEEE CS Press, 2005, pp. 2-9.
16. D.A. Bader et al., ''BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture on Bioinformatics Applications,'' *Proc. Int'l Symp. Workload Characterization* (WWC 05), IEEE CS Press, 2005, pp. 163-173.

vices currently lag behind by at least several orders of magnitude.

The second category of computational crudeness is best illustrated by the trend of loose decoupling between the control and I/O devices. For many bioimplantable devices, only simple analog sensors or actuators are implanted within the human body; the critical control and processing components (which are usually bulky and tethered) are left outside the body. This detachment of the computational control and processing unit from the sensor or actuator peripherals can leave the entire implant device vulnerable to many potential malfunctions that can considerably hinder its effectiveness, efficiency, and usability.

To address the computational crudeness of bioimplantable devices, we must answer two important research questions: First, how can we increase total computing power, so that a bioimplantable device not only meets basic functional control and processing demands but also addresses critical issues such as reliability, security, and upgradability? Second, how can we integrate this increased level of computing power into the bioimplantable device while still satisfying the more constrained design requirements associated with this stringent implant platform, such as power, energy, size, thermal characteristics, and bio-compatibility? Answering these two questions requires a paradigm shift; we need to holistically redesign the processor and accelerator architectures we know today into forms better suited to this high-impact, high-reward domain.

Designing processor and accelerator architectures for modern computer systems relies heavily on the use of benchmarks. This design process typically involves making balanced trade-offs between the desired system performance characteristics (speed, functionality, security, reliability, and so on) and costs (power, area, time to market, nonrecurring engineering, and so on). The designer does this by iteratively "walking through" the design space constituted by the workload characteristics of the target benchmarks and the different system configurations and design options to be explored. As a result, this benchmark-driven architecture innovation process has made the selection of proper benchmarks a critical part of the overall system design flow.

One of the biggest obstacles to architecture innovation in bioimplantable computing is that there is currently no compilation of adequate workloads that can best represent the requirements of this emerging domain. Indeed, a variety of benchmarks have been proposed in the past, including Dhrystone, Linpack, Whetstone, MediaBench, MiBench, and, the most widely used, the SPEC CPU benchmark suites. (The "Related Work on Benchmarks" sidebar discusses these and others.) Unfortunately, most of these benchmark suites target traditional desktop and laptop, scientific, embedded, or multimedia systems. Although they have provided good vehicles for exploring conventional computer system research, they do not offer representative and significant biomedical workloads to capture the essential evaluation milestones to guide architecture innovation for future bioimplantable systems. Without this representative bioimplant benchmark suite, it is practically infeasible to make the appropriate performance and cost trade-offs.

The ImplantBench initiative is our first attempt at filling this gap. Unlike traditional benchmark suites that aim to establish the gold-standard set of applications for which new architectures must be optimized, ImplantBench is a workload taxonomy that aims to provide a higher level of abstraction for reasoning about bioimplantable workload requirements. Our goal is to delineate workload requirements in a manner that is not overly specific to individual applications, so that we can draw broader conclusions about architecture requirements for bioimplantable processors and accelerators. Thus, we propose a set of computational kernels, each of which captures a pattern of computation common to a class of important applications that either already exist or most likely will appear in the domain of bioimplantable computing. To make our workload taxonomy approach more concrete and tangible, we also provide some representative code examples for each class of computational kernel. These codes are examples rather than standards; we provide them mainly to reach out across the computer architecture community and broaden participation in the field. Our projection is that many future bioimplantable applications will combine these computational kernels in various ways.

In this article, we describe our comprehensive study, characterization, compilation, and projection of a representative set of open-source algorithms and workloads that are currently in use or will most likely be adopted by future bioimplantable systems. We have undertaken the necessary technical tasks of converting, rewriting, implementing, and debugging the ImplantBench workloads to enhance their portability from various platforms into the widely adopted Linux and ×86 research platforms. The benchmark suite we propose can be conveniently accessed and used by researchers within and across all relevant disciplines, free of charge.

We also conducted a realistic case study that captures the essential workload characteristics of the included computational kernels and demonstrates their usage with the popular GNU development infrastructure and SimpleScalar tool chains. Our insights and findings from the case study suggest important design criteria that architects should consider when designing processor and accelerator architectures for emerging bioimplantable systems.

ImplantBench components and descriptions

All ImplantBench applications are coded in high-level standard C programming language and are thus compilable in most compiling platforms. In addition, we have published all the applications as either open-source code or under a general public license (GPL), which makes this benchmark suite easily and publicly available to a wide user community.

ImplantBench consists of 23 specific applications grouped into six important categories: security, reliability, bioinformatics, genomics, physiology, and heart activity. We based this selection on how commonly categories of applications are used in academia, and how likely they are to be used in the future. Moreover, ImplantBench is flexible enough to be adjusted and improved when new applications become available and extendable. Table 1 lists the main categories and corresponding benchmark components, which we explain in more detail in the following sections.

Table 1. Categories and benchmark components in ImplantBench.

Category	Benchmark components
Security	Haval, Khazad, SHA2
Reliability	CRC-CCITT, Luhn, Hamming, Reed-Solomon, Alder-32
Bioinformatics	ELO, LM-GC, UPGMA, Unger-Moult
Genomics	Jensen-Shannon, HMM, Nussinov-Jacobson, Smith-Waterman
Physiology	AFVP, EVAL_ST, ECGSYN, RRGen
Heart activity	Activity, pNNx, CO

Security

When smart chips are implanted in the human body for clinical purposes, certain sensitive medical information may need to be transferred secretly—safe from inadvertent invasion or from deliberate access by unauthorized parties. Thus, security becomes the first important characteristic with which bioimplant researchers should be concerned.

Haval (hash algorithm with variable length of output) is a one-way hashing algorithm that supports 15 different levels of security.[1] Haval can produce hashes of different lengths in 128, 160, 192, 224, and 256 bits, and also lets users specify the number of rounds to be used to generate the hash. The algorithm is very efficient and particularly suited for real-time applications.

Khazad is an iterated 64-bit block cipher with 128-bit keys.[2] It comprises eight rounds: Each round consists of eight byte-to-byte S-box parallel lookups, a linear transformation (multiplication by a constant MDS diffusion matrix), and round-key addition. (MDS is an Internet authentication protocol.) Khazad makes heavy use of involutions as subcomponents; this minimizes the difference between the algorithms for encryption and decryption, making its hardware implementation more unified.

SHA2 functions are used to generate a condensed representation of a message called a message digest, suitable for use as a digital signature.[3] The SHA2 functions are considered more secure than those of SHA1, with which they share a similar interface. For this benchmark suite, we chose SHA-256 from the SHA2 function family, which includes SHA-224, SHA-256, SHA-384, and SHA-512.

Reliability

For chips implanted in the human body, data communication should rely chiefly on wireless techniques rather than wired connection, which is impractical and awkward in this context. Information communicated from and to the implanted processors must be checked and confirmed as correct. Thus, our benchmark suite integrates simple but also effective reliability algorithms.

Cyclic redundancy check (CRC) is a type of hash function that has the ability to detect errors caused by data transmission. The check increases the overall reliability of data transmissions to ensure that data being streamed has not been altered or misread. CRC-CCITT, an algorithm standardized by the International Telephone and Telegraph Consultative Committee (CCITT), is preferred over many other error-checking algorithms because of its easy realization in binary hardware.[4]

The *Luhn algorithm*, also known as the "mod 10" algorithm, is a simple checksum formula used to validate a variety of identification numbers.[5] It was created by IBM scientist Hans Peter Luhn and is described in US patent 2,950,048. Its low hardware complexity makes it suitable for low-power implementations.

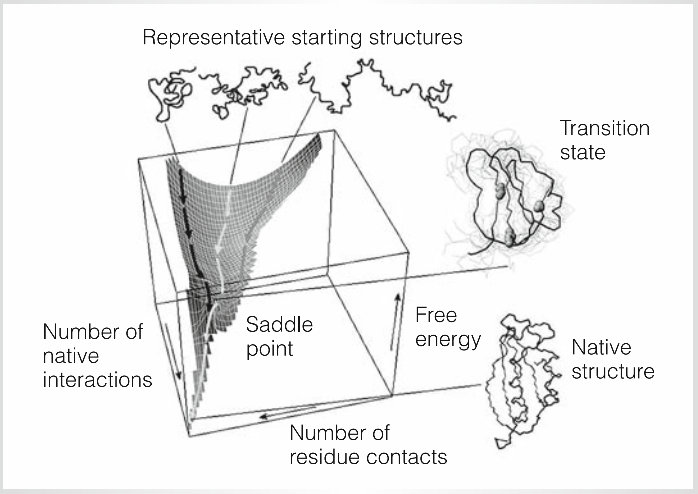
Representative starting structures
Transition state
Number of native interactions
Saddle point
Free energy
Native structure
Number of residue contacts

Figure 1. Schematic energy landscape for protein folding (courtesy of Christopher M. Dobson and *Nature*).[12]

Hamming code is another error-checking and -correcting algorithm.[6] Useful for information streaming between nodes, Hamming codes come in a variety of forms. The algorithm works for several different bit counts. It operates by using a varied parity bit; when a parity bit is incorrect upon reading, a flag is thrown. The error is either simply identified, or it is corrected as well.

Reed-Solomon error correction is an error-checking and -correcting algorithm that uses a polynomial to keep track of data.[7] Similar to other error-checking and -correcting algorithms, Reed-Solomon code is useful for data transmission and data storage and recovery.

Adler-32, invented by Mark Adler, is a checksum algorithm for transmission error detection.[8] Compared to a CRC of the same length, it trades reliability for speed. Adler-32 uses relatively simple processes to check data integrity. It was designed to overcome some of the inadequacies of simply summing all the bytes, as the original checksum did.

Bioinformatics

As the genomics field progresses, having accessible computational methods with which to extract, view, and analyze genomic information becomes essential. Bioinformatics lets researchers sift through the massive biological data and identify information of interest. Algorithms and applications in this new computational field of biology are now expected to be used in vivo with limited computing resources as enhanced diagnosis assistance methods.

Evolution simulation algorithm (ELO) is an implementation of Dawkins' program, which simulates evolution of strings and counts the number of generations necessary to converge to a given target string.[9,10] The algorithm forms mutant copies of a source string by changing each of its characters with a certain probability, and then replaces the source with one copy that agrees most with the target string.

Lagrange multipliers for optimality of genetic code (LM-GC) is a program that uses Lagrange multipliers, a method for finding the extrema of a function of several variables subject to one or more constraints, to estimate the optimization level of standard genetic code—an approach first taken by DiGiulio.[10] It creates a distance matrix, whose (i, j)th entry is the fourth power of the difference between polar requirement of the ith amino acid and polar requirement of the jth amino acid.

The clustering algorithms program suite implements variants of the *unweighted pair group method with arithmetic mean* (UPGMA) clustering algorithm for phylogeny trees.[10,11] This program suite allows the computation of both UPGMA and WPGMA (unweighted and weighted forms) and also implements the Farris transformed distance method. UPGMA is a straightforward method of tree construction, with the original purpose of constructing taxonomic phenograms. Widely adopted in the study of bacterial population biology, the method uses a sequential algorithm, in which local homology between operational taxonomic units (OTUs) is identified in order of similarity.

The *Unger-Moult protein-folding algorithm* models the physical process by which a polypeptide folds into its characteristic 3D structure, shown in Figure 1.[10] Unger-Moult is a genetic algorithm (with hybrid Metropolis steps) implemented by Schubert and Buchetmann, for folding a protein on a

2D lattice to maximize the number of hydrophobic-hydrophobic (HH) contacts at unit distance (http://www.cs.bc.edu/~clote/ComputationalMolecularBiology/ungerMoult.c).

Genomics

Genomic-related programs are usually considered computation-intensive tasks. However, some simple and preliminary genomic analysis could be shifted to implantable devices and implemented to serve real-time or time-urgent therapeutic applications that automatically adapt to changing conditions.

Hidden Markov models (HMMs) are statistical models that were initially developed for speech recognition, and have subsequently been used in numerous biological-sequence analysis applications.[10,13] Current applications of HMMs in computational biology include protein families modeling, gene finding, transmembrane helices prediction, tertiary structure prediction, and others. An HMM models a biological sequence—for example, a protein, DNA, or RNA sequence—as an output generated by a stochastic process progressing through discrete time steps. At each time step, the process outputs a symbol (an amino acid or a nucleotide) and moves from one of a finite number of states to the next state.[14]

Jensen-Shannon divergence (JSD) is a measure of the "distance" between two probability distributions, which can also be generalized to measure the distance (similarity) between a finite number of distributions.[10,15] JSD is a natural extension of the Kullback-Leibler divergence (KLD) to a set of distributions. This program computes the Jensen-Shannon divergence $D(U, V)$ for a segmentation of the whole genome W into left segment U and right segment V.

A common problem for researchers working with RNA is determining the 3D structure of a molecule, given just the nucleic acid sequence. The *Nussinov-Jacobson algorithm* (N-J) uses dynamic programming and backtracking to determine the optimal RNA secondary structure by finding the structure with the maximum number of base pairs.[10,16] Here, base pair means Watson-Crick or guanine-uracil (GU) base pair varied in the code.

Smith-Waterman (S-W) is a well-known algorithm for performing local sequence alignment;[10,17] that is, it determines similar regions between two nucleotide or protein sequences. Instead of looking at the total sequence, the S-W algorithm compares segments of all possible lengths and optimizes the similarity measurement. Besides implementing Smith-Waterman local sequence alignment using similarity matrices, this S-W program also implements global sequence alignment (Needleman-Wunsch, a global alignment on two sequences proposed by Needleman and Wunsch in 1970[18]).

Physiology

One of the most important applications of implantable chips is to collect and analyze intrinsic physiological signals—for example, electrocardiogram (ECG or EKG), electromyogram (EMG), and electroencephalogram (EEG) signals. Many of these applications rely heavily on advanced signal-processing techniques to break these signals into segments to facilitate abnormality detection. For instance, a typical ECG tracing of a normal heartbeat can be decomposed into a P wave, a QRS complex, and a T wave. (The letters P, Q, R, S, and T were originally assigned to the various deflections by Willem Einthoven, a Nobel Prize Laureate in physiology and medicine.) Another emerging research area is to synthesize and simulate such intrinsic physiological signals for artificial organs and prosthetics.

Atrial fibrillation and ventricular pacing (AFVP) is a realistic ventricular rhythm model of atrial fibrillation (AF), generating a synthesized beat-to-beat interval sequence of ventricular excitations with a realistic structure.[19] The model treats the atrial ventricular junction (AVJ) as a lumped structure characterized by refractoriness and automaticity. Bombarded by random AF impulses, the AVJ can also be invaded by the ventricular-pacing-induced retrograde wave. The model also considers electrotonic modulation by blocked impulses. With proper parameter settings, the model can

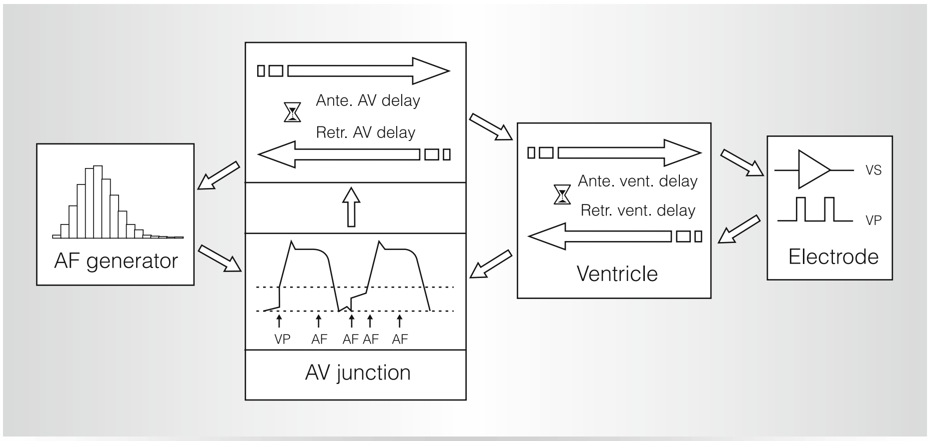

Figure 2. Schematic flow of AFVP model (courtesy of Jie Lian).

account for most principal statistical properties of the intervals between heartbeats (known as *RR intervals*) during AF. Figure 2 illustrates the AFVP flow.

The *EVAL_ST* program was developed to evaluate and compare the performance and robustness of transient ST-episode detection algorithms, as well as to predict real-world clinical performance and robustness.[20] (An ST episode is a transient ST-segment depression or elevation of at least 0.10 mV.) It provides first-order (record-by-record) and second-order (aggregate, gross, and average) performance statistics for evaluation and comparison of transient ST-episode detection algorithms. EVAL_ST lets us assess an algorithm's accuracy in

- detecting transient ST episodes,
- distinguishing between ischemic and nonischemic heart-rate-related ST episodes,
- measuring durations of ST episodes and ischemic ST episodes, and
- measuring ST segment deviations.

The *ECGSYN* program generates a synthesized ECG signal from user-settable mean heart rate, number of beats, sampling frequency, waveform morphology (timing, amplitude, and duration of P, Q, R, S, and T), standard deviation of the RR interval, and LF/HF ratio (a measure of the relative contributions of the low- and high-frequency components of the RR time series to total heart rate variability).[21] Using a model based on three coupled ordinary differential equations, ECGSYN reproduces many of the features of the human ECG, including beat-to-beat variation in morphology and timing, respiratory sinus arrhythmia, QT dependence on heart rate, and R-peak amplitude modulation.

Physicians have long understood that a metronomic heart rate is pathological, and that the healthy heart is influenced by multiple neural and hormonal inputs that result in variations in heart interbeat (RR) intervals, at time scales ranging from less than a second to 24 hours. Given information about heart rate variability, *RR Generator* (RRGen) was developed to simulate a realistic sequence of RR intervals.[22] Although the intricate interdependencies of variations exist at different scales, RRGen tries to create a simulation that is sufficiently realistic to mislead an experienced observer to some extent. Figure 3 shows a 10-hour time series of RR intervals.

Heart activity

Cardiovascular disease claims more lives yearly than cancer does. Thus, we selected three representative programs that record, analyze, evaluate, and diagnose heart activity, and we set a separate category for cardiologic-relevant applications.

Activity, a tool for activity estimation from instantaneous heart rate, was developed for deriving an "activity index" based on measurements of mean heart rate, total power of the instantaneous heart-rate time series over a given interval, and stationarity.[23] Using only a heart-rate time series, it is possible to measure a number of features that reflect the level of physical activity. The algorithm developers tested Activity using a set of 35 ECG recordings for which an independent activity indicator is available. It consistently selects periods of minimum activity that are in agreement with the independent activity indicator.

The pNN50 statistic is a time-domain measure of heart-rate variability (HRV), where pNN50 count is defined as the mean number of times per hour in which the change in consecutive normal sinus (NN) intervals exceeds 50 milliseconds. The *pNNx heart rate variability metric* was proposed to help assess parasympathetic (vagal) activity from 24-hour ECG recordings.[24,25] It has proved very useful in providing diagnostic and prognostic information in a wide range of conditions.

Cardiac output (CO), defined as the volume of blood pumped by the heart per unit time (often expressed in liters per minute), is the critical variable characterizing circulatory function. However, it is also one of the most difficult to measure. By introducing a small amount of cool fluid into the blood upstream of the heart and observing how rapidly the blood temperature equilibrates, we can calculate the flow and thus the CO. The *cardiac output estimation from arterial blood pressure waveforms* program implements 11 improved CO estimation algorithms and compares them with thermodilution cardiac output (TCO), a gold standard among CO measurements.[26]

Experimental infrastructure

We assembled the programs and algorithms we have just described from various public sources and readied them for use in ImplantBench primarily in three ways:

- We gathered the widely used, open-source programs—such as AFVP, ECGSYN, and pNNx—under the GNU General Public License.
- For applications that existed only as algorithms, such as Alder-32 and Luhn, we implemented them in standard C language.

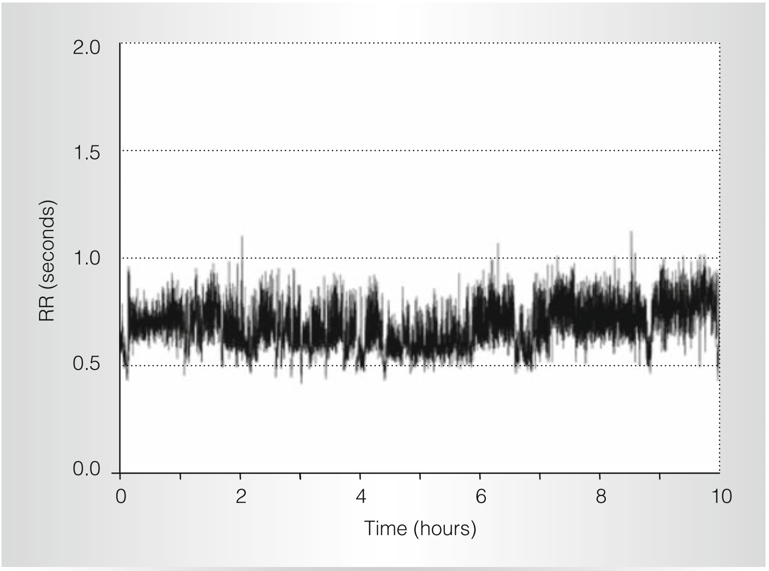

Figure 3. Example 10-hour time series of RR intervals. The range of heart rate is roughly 70 to 120 beats per minute (courtesy of George Moody and *PhysioNet*; http://www.physionet.org/challenge/2002).

- In cases in which the implementations didn't meet distribution quality or exist in other high-level implementations (Matlab, C++, Java), we reimplemented the algorithms in standard C. The CO program is an example.

To perform our workload characterization and analysis, we cross-compiled all the benchmark programs in our ImplantBench suite into SimpleScalar PISA binaries using GCC 2.7.2.3 on both a Fedora 7 Linux system and a Cygwin platform on Windows with −O2 optimization enabled. With slight modifications to header files and makefiles, all of the benchmark programs compiled properly. We simulated all generated PISA binaries on the SimpleScalar simulator infrastructure for fast, flexible, and accurate cycle-level system simulation (http://www.simplescalar.com).

To gather instruction distribution, branch class, address mode, and segment profiling information, we used *sim-profile*. We gathered cache performance data by simulating the memory references of all the benchmark applications with *sim-cheetah*, which can simulate multiple cache configurations in a single pass. We used *sim-outorder* to simulate and analyze branch

Table 2. Summary of ImplantBench execution commands.

Application	Execution command
CRC-CCITT	crc [options] <filename>
Luhn	luhn <input value strings>
Hamming	Hamming <filename>
Reed-Solomon	rs [options] <filename>
Alder-32	Alder <filename>
Haval	haval [-cl -el -s] <filename>
Khazad	khazad <number>
	khazad [options] <filename>
SHA2	sha [options] <filename>
ELO	evolution <strings> <number of generations>
LM-GC	digiulio <polarity_vector_file> [v1lv2lv3lv4lv5]
UPGMA	multipleGotoh <filename>
	phy5 <filename> -t/u/w
Unger-Moult	ungerMoult <filename>
HMM	markov <filename>
	hmm2 <number>
Jensen-Shannon	jensenShannon <filename>
Nussinov-Jacobson	rnaMaxNumBasePairs RNAnucleotideString
Smith-Waterman	smithWaterman protein1 protein2 PAM gapPenalty
AFVP	afvp <config_file> <random number seed>
EVAL_ST	eval_st –r <database>_<algorithm>.evl <record> -[ils]
	eval_st –a <database>_<algorithm>.evl –[ils] [-el-b [NR_TRIALS]]
ECGSYN	ecgsyn [options] <input values>
RRGen	rrgenv3 seed [tmax] P<ectopy> P<noise>
Activity	activity [options] <'len' values> <instantaneous heart rate file>
pNNx	pNNx –r <record name> -a <annotator name> pnnlist < <interval list>
Cardiac output	evco <input file>

predictions and dynamic execution performance. We will distribute the detailed configuration information along with the ImplantBench release.

As for the use of ImplantBench, Table 2 provides an instruction summary for all ImplantBench benchmarks, including the command line executables and their arguments. We will provide the detailed command line usage in the source distribution.

Results and insights

Here, we offer both high-level qualitative analyses and detailed quantitative evaluation to characterize the ImplantBench workloads via cycle-level system simulation.

Benchmark component analysis

Because we used the SimpleScalar simulator to simulate the entire ImplantBench suite, it is feasible to gather the full dynamic instruction trace cycle by cycle. For our benchmark program simulation, we used moderate input data sets, striking a balance between dynamic runtime and how well the data sets represent real workloads. Table 3 shows that—except in the reliability group—the total number of dynamic instructions simulated for the ImplantBench programs are in the range of hundreds of millions, with a maximum greater than two billion dynamic instructions.

In the reliability group, because bioimplantable systems should be ultralow-power and energy efficient due to the extremely limited energy provided by either batteries or scavenge energy, our design objective is to maintain the necessary reliability as much as possible with minimal power consumption overhead. That is, we attempted to select useful and efficient algorithms with minimum energy consumption requirements. We chose the programs for the reliability category from a large pool of candidates, some of which might offer better protection than those included. The primary reason for not including such other programs at this point is that the power requirement they impose is simply larger than we can afford. Therefore, the total number of dynamic instructions simulated for each respective category is sufficient for the purpose of our workload characterization analyses.

Table 3. Benchmark program sizes.

Category	Benchmark program	Dynamic instructions (no.)
Reliability	CRC-CCITT	12,231
	Luhn	6,333
	Hamming	984,576
	Reed-Solomon	767,125
	Alder-32	8,995
Security	Haval	430,413,126
	Khazad	14,355,628
	SHA2	428,459,652
Bioinformatics	ELO	48,173,094
	LM-GC	1,849,203
	UPGMA	3,928,748
	Unger-Moult	2,475,136,094
Genomics	Jensen-Shannon	57,432,567
	HMM	37,471,844
	Nussinov-Jacobson	473,967
	Smith-Waterman	58,139,042
Physiology	AFVP	246,568,756
	EVAL_ST	414,258,851
	ECGSYN	1,813,104,873
	RRGen3	87,291,436
Heart activity	Activity	441,153,291
	pNNx	246,456,300
	CO	387,740,186

Instruction class distribution analysis

There are four main classes of instructions: memory (load and store), integer operations, floating-point operations, and control (unconditional and conditional branches). Control-intensive applications spend many more operations on branch control, and correspondingly execute more branch-related instructions. Similarly, in I/O-intensive applications, frequent data access and transmission from memory are dominant in all operations, so load and store instructions occupy the largest portion in the instruction distribution. Computation-intensive programs have a larger percentage of integer or floating-point ALU operations for specific applications. Figure 4 shows the instruction distribution of all the ImplantBench programs.

This figure demonstrates some distinctive characteristics of the benchmark suite. The security benchmarks spent more than 70 percent of their total runtime in executing integer computations. This is because most of the processing required for secure encryption and decryption consists of repeated computational operations on the same set of integer data. However, security benchmarks also have the least branch control operations among all the programs, because of their relatively easier program control structure.

Compared with the security programs, the reliability benchmarks have more load and store operations: Memory access operations account for approximately 40 percent of all executed instructions. This is because most of these programs must frequently obtain an input data stream from memory, perform data integrity checks and corrections, and then write results back to memory. Both the security and reliability categories have very few floating-point operations, because their emphasis is on processing integer data.

On the other hand, the floating-point instructions in the heart activity and physiology groups have a stronger presence. Most programs in these categories must sample or process ECG data, heart activity data streams, and blood pressure waveforms, which typically are nonintegral numeric data. The different instruction distributions of these benchmark programs reflect a strong correlation to their real-world applications in the biomedical field.

Figure 4 also brings to light the similar instruction distributions among all programs within the respective categories. For example, the three programs in heart activity category—Activity, pNNx, and CO—have a very similar instruction distri-

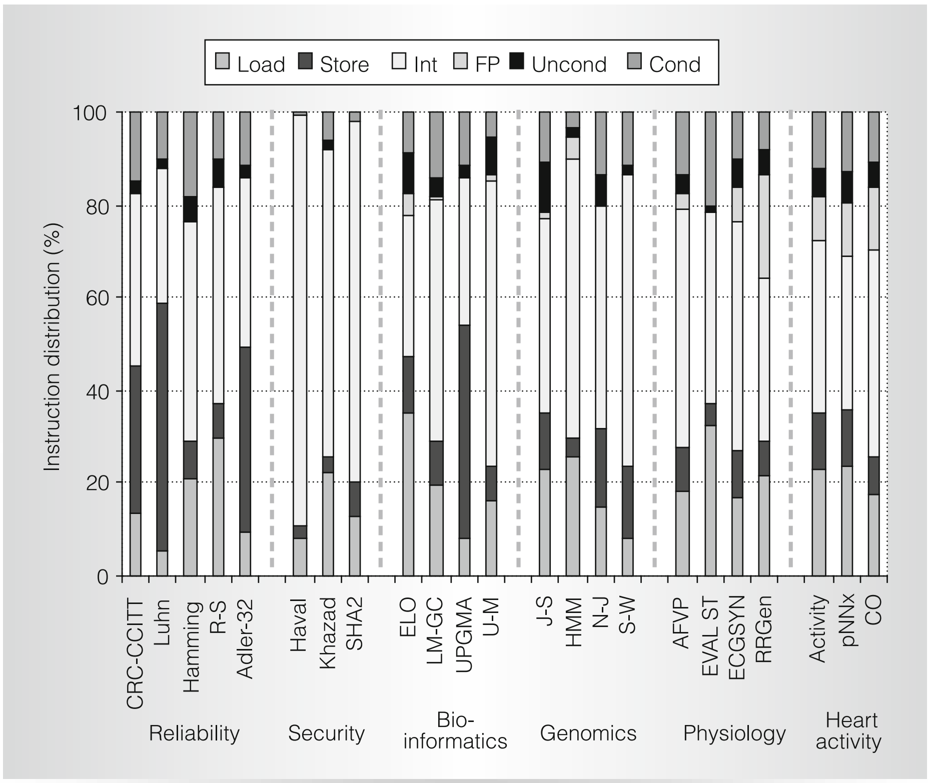

Figure 4. Dynamic instruction distribution.

bution. This intragroup similarity applies to the other four groups as well. In contrast, when we consider all the programs from the entire benchmark as a whole, they show great variation in instruction distribution. For example, across the entire suite, the integer instructions range from 30 percent to nearly 90 percent. Load and store operations also vary from 10 percent to more than 50 percent, and so on. The obvious variation between programs from different categories and the notable similarity among programs within the same category convince us that the selection and grouping strategies we used to compose ImplantBench are reasonable and logical.

Branch analysis

ImplantBench also shows a great variation in program control flow. First, let's look at the dynamic branch class distribution, shown in Figure 5. This figure shows the distinct similarity among programs within their respective categories. Additionally, it shows that conditional direct branch operations are dominant among all the branch types, taking up at least 40 percent of total branch instructions executed across the entire benchmark suite, and for some programs as much as 90 percent. The three other significant branch types are unconditional direct mode, call direct mode, and unconditional indirect mode. These three modes together account for approximately 40 percent of total branch instructions. The remaining two modes—conditional indirect and call indirect—are rarely used by the majority of the ImplantBench programs.

Another important issue we considered is the effectiveness of different branch prediction schemes. How well program branches can be predicted is an important criterion in modern computer architecture design. We used four schemes provided by SimpleScalar to simulate and compare branch prediction efficiency: a not-taken prediction scheme, an 8-Kbyte gshare two-level adaptive predictor, an 8-Kbyte bimodal predictor, and an 8-Kbyte combined bimodal and two-level predictor. We configured all the

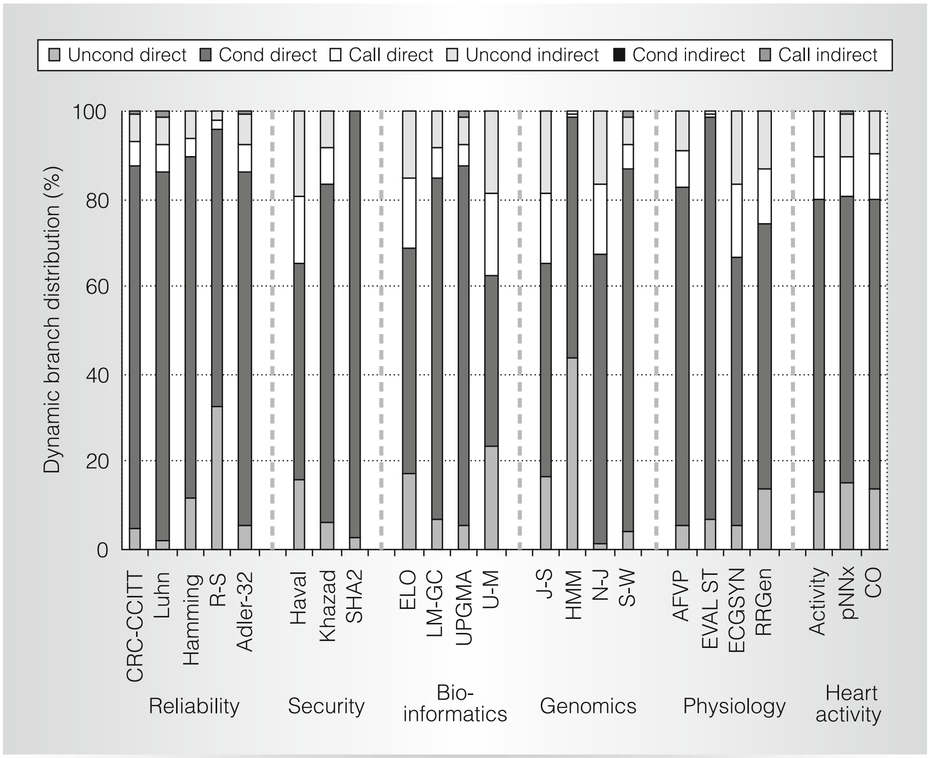

Figure 5. Branch class distribution.

predictors, except the not-taken strategy, with a 2-Kbyte branch target buffer (BTB). Figure 6 shows the detailed misprediction rates, expressed in mispredictions per thousand instructions. Increasing the sophistication of the predictor strategy results in a remarkable decrease in misprediction rates: a rate of up to 400 mispredictions per 1,000 instructions for the simple, not-taken predictor can be reduced to as few as 10 mispredictions per 1,000 instructions.

As for the remaining three predictor schemes, there is no obvious difference among their effects on improving branch prediction rates. Like SPEC, MiBench, and MediaBench benchmarks, ImplantBench has well over a 90 percent prediction success rate, and most branches in every benchmark program are predictable. However, some programs still have a slightly higher misprediction rate, such as those in the genomics and reliability categories. This is due to massive relative random data stream that these applications must transfer and process. In contrast, the security group has a slightly lower misprediction rate than the other categories, which is a result of the relatively more regular data-access pattern of these programs' integer-intensive operations.

Address mode analysis

Address mode is another important criterion we need to inspect. Figure 7 provides a detailed distribution for every address mode, including constant (const), global-pointer with constant (gp + const), stack-pointer with constant (sp + const), frame-pointer with constant (fp + const), general-purpose register with constant (reg + const), and general-purpose register with general-purpose register (reg + reg). These six addressing modes can be classified into three categories: constant only (const), register plus displacement with pre or post increment or decrement (gp, sp, fp, and reg + const), and registers only (reg + reg).

Figure 7 shows that the displaced addressing mode shows benchmark-wide dominance. Examining the displaced addressing mode in greater detail, Figure 7 shows that the order of dominance is reg +

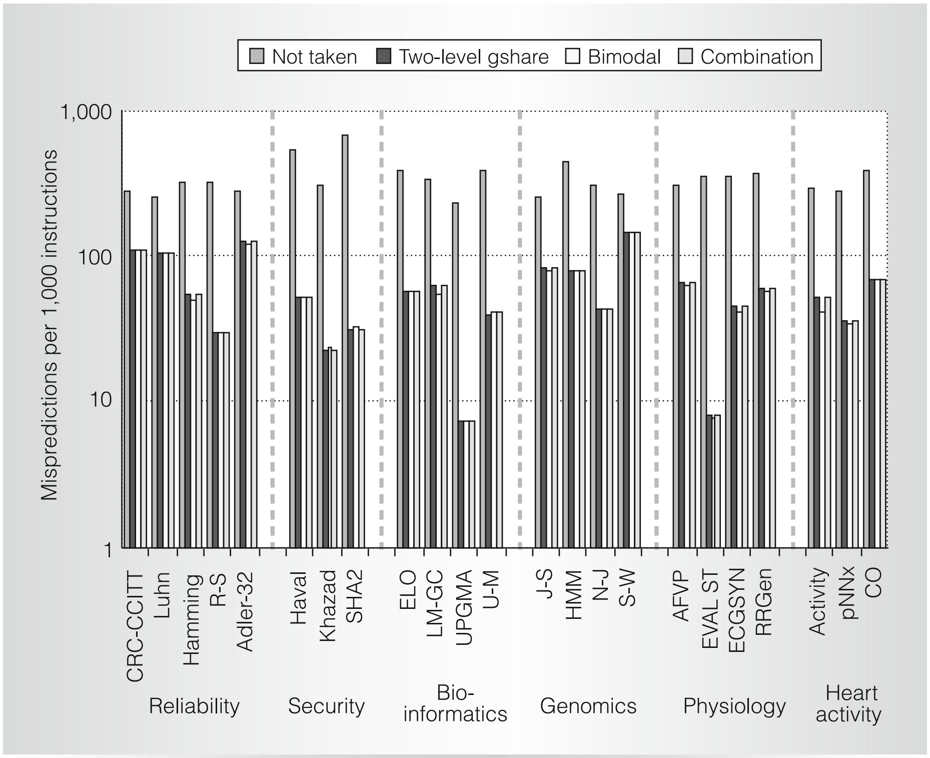

Figure 6. Branch prediction rates for several schemes.

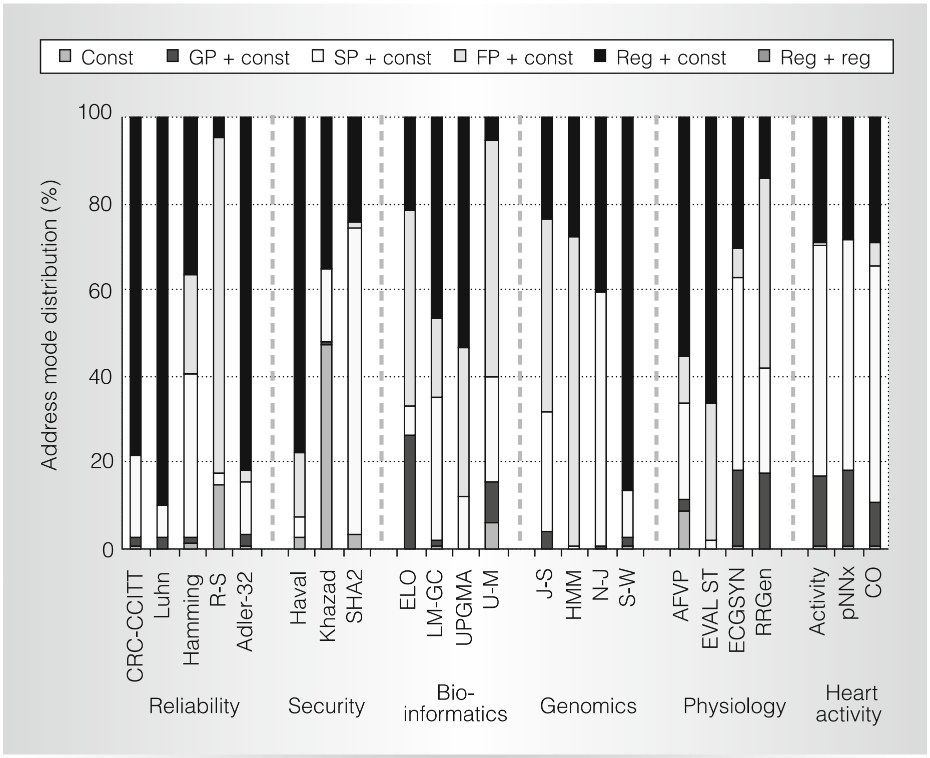

Figure 7. Address mode distribution.

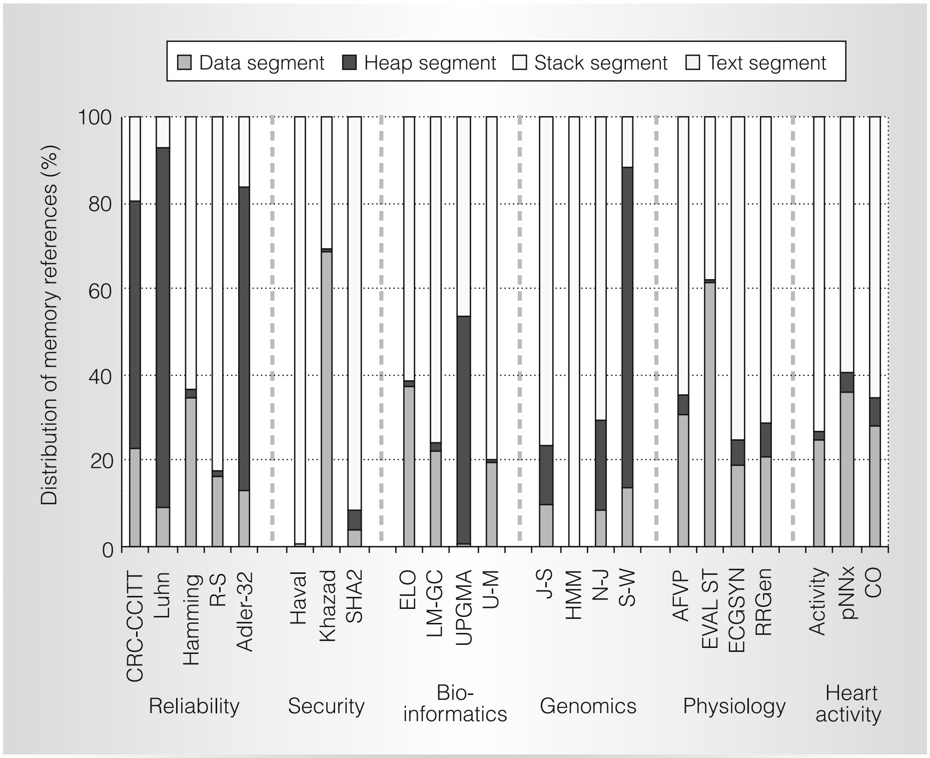

Figure 8. Load and store segment distribution.

const, followed by sp + const, and then fp + const—with some variation. Programs such as CRC-CCITT, Adler-32, Haval, and Smith-Waterman have a strongly dominant use of reg + const mode, with up to 90 percent of address mode use. On the other hand, fp + const dominates address mode use (up to 70 percent) in Reed-Solomon, Unger-Moult, and HMM. Khazad, which mainly uses the const address mode, is an exception within the suite.

The dominance of displaced address mode use, as shown in Figure 7, indicates the potential of using two-address operands instead of the common three-address operands when designing the instruction set architecture for bioimplantable applications.

Load and store segment analysis

A segment is a section of a program in an object file or in memory that contains the global variables that are initialized by the programmer. A program has four basic memory regions: data, stack, heap, and text. A data segment relates to the amount of memory used to store initialized data. A heap segment relates to the amount of memory currently being used for data created at runtime. A stack segment means the amount of memory used by the currently executing procedures or function calls and all the subroutines that have been invoked. A text segment refers the amount of memory used to store the actual machine code instructions. Thus, to understand the memory access characteristics of bioimplantable device applications, it is useful and significant to explore the segment distribution of all the memory references (loads and stores) in ImplantBench. Figure 8 gives the distribution details.

From Figure 8, we could conclude that benchmark programs with considerable runtime allocated to intermediate- and final-result data that must be written back to the main memory—such as CRC-CCITT, Luhn, Adler-32, UPGMA, and Smith-Waterman—have the biggest distribution in the heap segment. Khazad and EVAL_ST have the most data to be initialized at the beginning of program execution, so they also have the largest data segments—about 60 percent. In general, stack segment use is significant for all the benchmark programs. The variation is due to different layers of nested function calls and the total number of procedure calls invoked. The percentage of stack segments ranges from 10 percent (Smith-Waterman) to almost 100 percent (Haval, which has the least initialized data as well as the least intermediate data created during execution). Again we find interesting similarities within certain categories. For example, physiology, heart activity, genomics (except S-W), and bioinformatics (except UPGMA) all show similar segment characteristics.

Memory analysis

Simulations of memory performance focus on both the instruction cache and data cache, with various cache sizes (from 256 bytes to 128 Kbytes) and associativity configurations (one-, two-, four-, and eight-way). We used the SimpleScalar built-in cache simulators sim-cheetah and sim-cache to measure the cache miss rates. To perform this detailed cache performance analysis, we randomly chose five programs—Jensen-Shannon, pNNx, Activity, Hamming, and Khazad.

Figure 9 shows instruction cache performance of the selected benchmark programs. Most miss rate values decrease to less than 1 percent with associativity greater than four-way and cache size greater than 8 Kbytes. The variety of inner functionalities associated with a proportional distribution of instructions including floating-point and integer calculation, load and store, and branch lead to a relatively higher instruction cache miss rate for pNNx and Activity, which are designed for cardio-activity measurement and monitoring. In the security group, Khazad's miss rate drops drastically to less than 0.3 percent when cache size reaches 2 Kbytes, because of its relatively simple and symmetric computation structure.

Figure 9 also shows that miss rates drop to less than 0.1 percent for every program we simulated when we used the eight-way associativity scheme. That is, the eight-way scheme provides sufficient degrees of associativity to effectively converge the miss rates to a satisfactory level. In comparison, miss rates for some of the SPEC2000 benchmarks don't fall below 2 percent until cache size reaches around 16 to 32 Kbytes—8 to 16 times larger than that required by ImplantBench for similar miss rates. Thus, ImplantBench provides more opportunities for designs in the ultralow-power domain than do traditional benchmark suites.

Figure 10 indicates the data cache performance for the selected benchmark programs. As the cache size and the associativity increase, all of the programs illustrate a trend of decreasing data cache miss rates similar to those of the instruction cache. On average, miss rates for the data cache converge faster than those for the instruction cache as the cache size and associativity increase. The miss rates for all but one of the selected benchmarks become negligible with either associativity greater than four-way or cache size greater than 4 Kbytes. Khazad's relatively higher miss rate comes from the large number of inputs and the randomness of test vectors. However, owing to the large number of iterative encryption units in the cipher, Khazad's instruction cache performance remains stable because of relatively higher locality. Compared with traditional general-purpose and scientific benchmarks, ImplantBench has more easily cacheable instructions and data, again providing opportunities for constrained biomedical implant applications.

Performance analysis

To assess the achievable target performance for bioimplantable processor design, we need to determine the available instruction-level parallelism (ILP) exhibited by the ImplantBench programs. This led us to perform program throughput analyses on the programs, measured in instructions per cycle (IPC). Figure 11 gives the results of our IPC simulation and analyses. The greatest IPC values come from security applications Khazad and SHA2, which achieve up to 3 IPC. This is not surprising, because these programs feature more ILP-rich integer ALU operations and fewer memory and program controls. The lowest IPC values are for UPGMA, AFVP, and

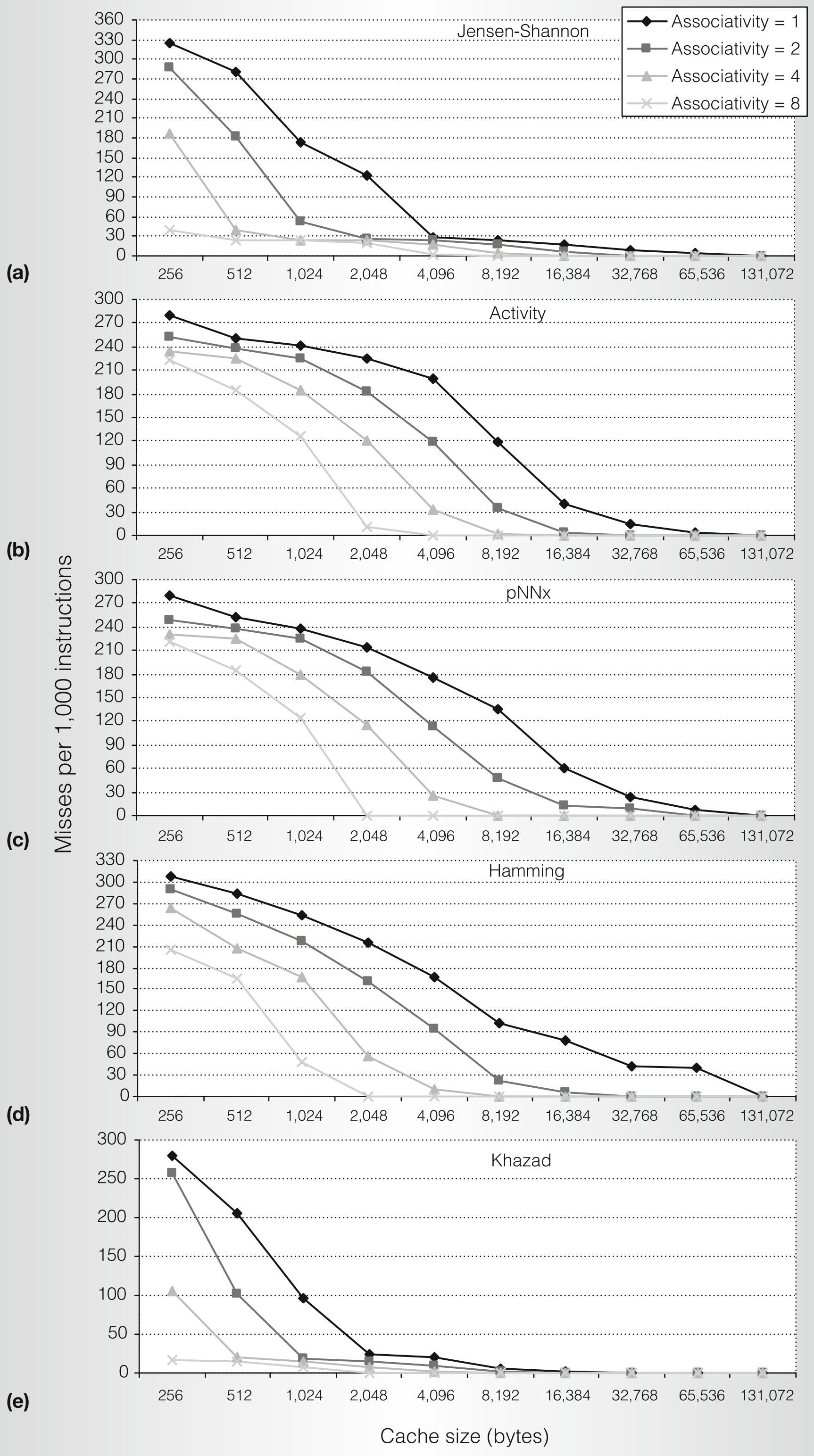

Figure 9. Instruction cache miss rates for various cache sizes and associativities.

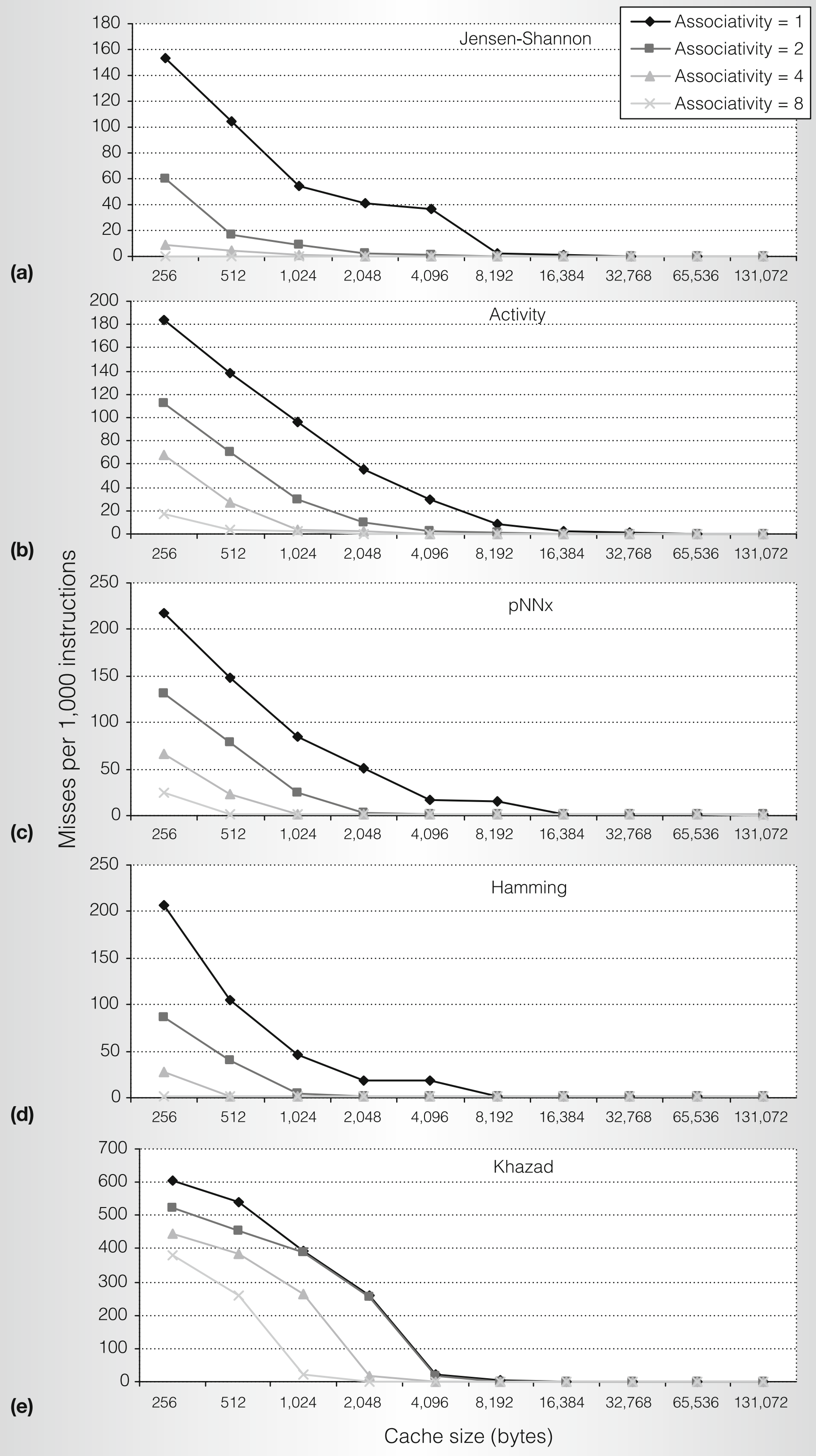

Figure 10. Data cache miss rates for various cache sizes and associativities.

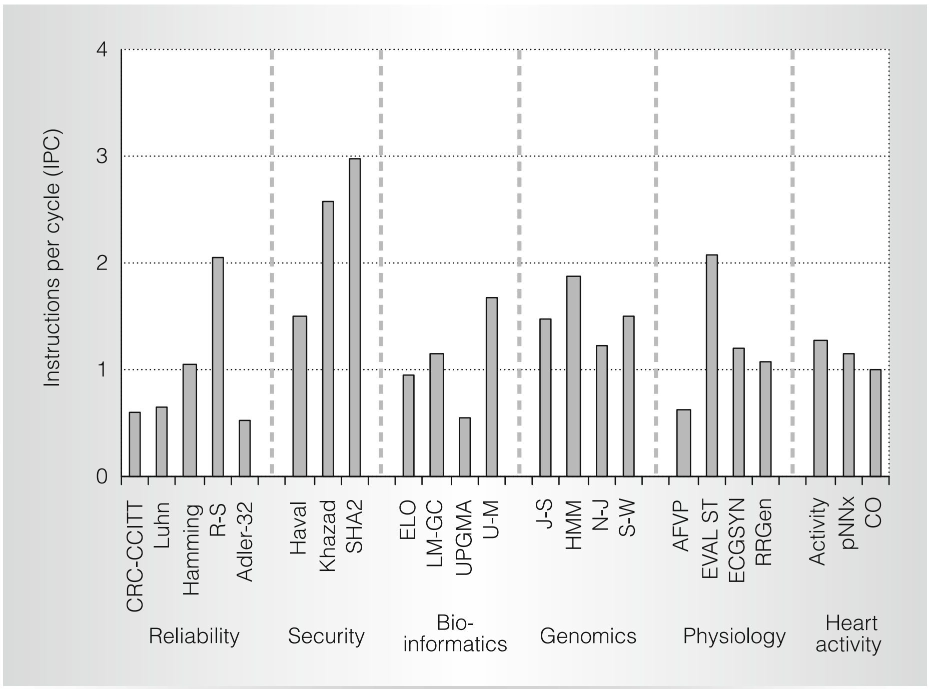

Figure 11. Program throughput for ImplantBench components, as measured by instructions per cycle (IPC).

Alder-32, which have many data dependencies and massive program flow-control changes. Most of the benchmark programs have IPC values greater than 1, which shows that there is some ILP available in ImplantBench, but it is not as abundant as it is in scientific or high-performance computing. Moreover, the high IPC variation suggests that ImplantBench programs are control and data intensive.

To the best of our knowledge, ImplantBench is the first benchmark suite of its kind that specifically focuses on establishing representative workloads to facilitate exploration of efficient and novel bioimplantable processor and accelerator architectures. If successful, these devices will be of enormous benefit to the high-impact biomedical augmentative and neuroprosthetics field.

Looking ahead, we hope to enhance our ImplantBench initiative in at least the following ways: We will incorporate different input data sets for each category, which will give the design under evaluation a wider range of dynamic runtime behavior (for example, small, medium, large, and very large dynamic instruction count), plus the common and worst-case energy and performance characteristics. We will also continue our current collaboration and outreach with physicians, scientists, and researchers from other relevant fields—such as neurological surgery, neuromuscular prosthetics, brain-computer interface, neuroscience, neurology, biophysiology, and biomedical engineering—to introduce new categories and expand existing ones to include more emerging workloads and algorithms.

We are currently in the process of preparing a public release of ImplantBench. The distribution will include the precompiled SimpleScalar PISA binary, C source code, and the reference input data set used in this study. We are striving to make the "out-of-zip experience" as smooth as possible. We expect that interested researchers will be able either to use the prebuilt binaries to run simulation or compile the provided source codes directly with any GCC-based or other mainstream C compilers.

With this article and the release of our first set of workload compilations, we invite the entire community to take the first stride in promoting human healthcare with innovative and cutting-edge bioimplantable computing technologies. MICRO

Acknowledgments

We thank our collaborators in the University of Pittsburgh School of Medicine—Robert Scalabassi and Mingui Sun in the Department of Neurological Surgery, and Chia-Lin Chang in the Department of Physical Medicine and Rehabilitation—for their proactive support and insightful consultation in helping us start this interdisciplinary bioimplantable computing initiative. We also thank other group members in the Advanced Computing Technology Laboratory—Yuan Sun for participating in initial workload surveying, porting, and analysis; and Timothy Sestrich for constructing and maintaining the ImplantBench website. We also gratefully acknowledge the original algorithm and application developers of the workloads included in this first version of ImplantBench. Finally, we thank the anonymous reviewers for their helpful feedback on improving this article.

References

1. Y. Zheng, J. Pieprzyk, and J. Seberry, ''HAVAL—A One-Way Hashing Algorithm with Variable Length of Output,'' *Proc. Int'l Conf. Cryptology: Advances in Cryptology*, LNCS 718, Springer-Verlag, 1993, pp. 83-104.
2. P. Barreto and V. Rijmen, ''The Khazad Legacy-Level Block Cipher,'' *1st Open NESSIE Workshop* 2000, pp. 13-14; https://www.cosic.esat.kuleuven.be/nessie/workshop.
3. H. Gilbert and H. Handschuh, ''Security Analysis of SHA-256 and Sisters,'' *Selected Areas in Cryptography*, LNCS 3006, Springer-Verlag, 2003, pp. 175-193.
4. R.N. Williams, *A Painless Guide to CRC Error Detection Algorithms*, Rocksoft Party, 1993.
5. H.P. Luhn, *Computer for Verifying Numbers*, US patent 2,950,048, Patent and Trademark Office, 1960.
6. D.J.C. MacKay, *Information Theory, Inference and Learning Algorithms*, Cambridge University Press, 2003.
7. S. Lin and D.J. Costello Jr., *Error Control Coding: Fundamentals and Applications*, 2nd ed., Prentice Hall, 2005.
8. P. Deutsch, A. Enterprises, and J.-L. Gailly, *ZLIB Compressed Data Format Specification*, RFC 1950, Internet Engineering Task Force, May 1996; ftp://ftp.rfc-editor.org/in-notes/rfc1950.pdf.
9. R. Dawkins, *The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe Without Design*, W.W. Norton & Co., 1996.
10. P. Clote and R. Backofen, *Computational Biology: An Introduction*, John Wiley & Sons, 2000.
11. H.C. Romeburg, *Cluster Analysis for Researchers*, Lifetime Learning Publications, 1984.
12. C.M. Dobson, ''Protein Folding and Misfolding,'' *Nature*, vol. 426, no. 6968, 18 Dec. 2003, pp. 884-890.
13. L.R. Rabiner, ''A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition,'' *Proc. IEEE*, vol. 77, no. 2, Feb. 1989, pp. 257-286.
14. T.M. Przytycka, ''Hidden Markov Models,'' *Nature Encyclopedia of the Human Genome*, D. Cooper, ed., Nature Publishing Group, 2003.
15. B. Fuglede and F. Topsoe, ''Jensen-Shannon Divergence and Hilbert Space Embedding,'' *Proc. Int'l Symp. Information Theory* (ISIT 04), IEEE Press, 2004, pp. 31-36.
16. R. Nussinov and A.B. Jacobson, ''Fast Algorithm for Predicting the Secondary structure of Single Standard RNA,'' *Proc. National Academy of Sciences*, vol. 77, no. 11, Nov. 1980, pp. 6309-6313.
17. T.F. Smith and M.S. Waterman, ''Identification of Common Molecular Subsequences,'' *J. Molecular Biology*, vol. 147, 1981, pp. 195-197.
18. S. Needleman and C. Wunsch, ''A General Method Applicable to the Research for Similarities in the Amino Acid Sequence of Two Proteins,'' *J. Molecular Biology*, vol. 48, 1970, pp. 443-453.
19. J. Lian, D. Mussig, and V. Lang, ''Computer Modeling of Ventricular Rhythm During Atrial Fibrillation and Ventricular Pacing,'' *IEEE Trans. Biomedical Engineering*, vol. 53, no. 8, Aug. 2006, pp. 1512-1520.

20. F. Jager, A. Smrdel, and R.G. Mark, ''An Open-Source Tool to Evaluate Performance of Transient ST Segment Episode Detection Algorithms,'' *Computers in Cardiology*, IEEE Press, 2004, pp. 585-588.
21. P.E. McSharry et al., ''A Dynamical Model for Generating Synthetic Electrocardiogram Signals, '' *IEEE Trans. Biomedical Engineering*, vol. 50, no. 3, Mar. 2003, pp. 289-294.
22. G.D. Clifford, F. Azuaje, and P.E. McSharry, *Advanced Methods and Tools for ECG Analysis*, Artech House Publishing, 2006.
23. G.B. Moody, ''ECG-Based Indices of Physical Activity,'' *Computers in Cardiology*, IEEE Press, 1992, pp. 403-406.
24. A.L. Goldberger et al., ''PhysioBank, Physio Toolkit, and PhysioNet Components of a New Research Resource for Complex Physiologic Signals,'' *Circulation*, vol. 101, no. 23, 13 June 2000, pp. e215-e220.
25. J.E. Mietus et al., ''The pNNx Files: Re-examining a Widely Used Heart Rate Variability Measure,'' *J. Heart*, vol. 88, no. 4, Oct. 2002, pp. 378-380.
26. J.X. Sun et al., ''Estimating Cardiac Output from Arterial Blood Pressure Waveforms: A Critical Evaluation Using the MIMIC II Database,'' *Computers in Cardiology*, IEEE Press, 2005, pp. 295-298.

Zhanpeng Jin is a PhD student of electrical and computer engineering at the University of Pittsburgh. His research interests include computer architecture; biomedical and bioimplantable computing systems; ultra low-power, energy-efficient processors; VLSI and ASIC system design; and mathematical modeling. He has a BS and an MS in computer science from the Northwestern Polytechnical University in China. He is a student member of IEEE and the IEEE Computer Society.

Allen C. Cheng is an assistant professor in the departments of Electrical and Computer Engineering, Computer Science, Neurological Surgery, and Neural Engineering in the McGowan Institute for Regenerative Medicine at the University of Pittsburgh, where he also serves as director of the Advanced Computing Technology Laboratory (ACT). His research interests include the interdisciplinary confluence of computer engineering, computer science, electrical engineering, neural engineering, biomedical engineering, and medicine. He has a BS in computer engineering from North Carolina State University, and an MS and a PhD in computer science and engineering from the University of Michigan, Ann Arbor. He is a member of the IEEE, the ACM, and the American Association for the Advancement of Science (AAAS).

Direct questions and comments about this article to Allen C. Cheng, Dept. of Electrical and Computer Engineering, University of Pittsburgh, 3700 O'Hara Street, Pittsburgh, PA 15261; accheng@ece.pitt.edu.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/csdl.