Title
Hardware Variability-Aware Embedded Software Adaptation

Author
Wanner, Lucas Francisco

Publication Date
2014

Peer reviewed|Thesis/dissertation

University of California
Los Angeles

Hardware Variability-Aware Embedded Software Adaptation

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Lucas Francisco Wanner

2014

Abstract of the Dissertation

Hardware Variability-Aware Embedded Software Adaptation

by

Lucas Francisco Wanner

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2014

Professor Mani B. Srivastava, Chair

With scaling of semiconductor fabrication technologies and the push towards deep sub-micron technologies, individual transistors are now composed by a small number of atoms. This makes it difficult to achieve precise control of manufacturing quality, with the added consequence that even slight differences in manufacturing can result in significant fluctuation in critical device and circuit parameters (such as power, performance, and error characteristics) of parts across the die, die-to-die and over time due to changing operating conditions and age-related wear-out. Instance and temperature-dependent variation, particularly in power consumption, has a direct impact on application quality and system lifetime for battery powered, energy constrained systems.

In this work we discuss software approaches to handle variability in power consumption of embedded systems. We characterize power variation in contemporary embedded processors, introduce tools for the evaluation of variability-aware software, and focus on two variability-aware software approaches: task activation control through variability-aware duty cycle scheduling and algorithmic choice.

We measured and characterized active and leakage power for a contemporary ARM Cortex M3 processor, and found that across a temperature range of 20–60°C

there is 10% variation in active power, and 14x variation in leakage power. As embedded processors in more advanced technologies become commonplace, the variations will increase significantly.

While contemporary hardware already suffers from variability, the evaluation of a variability-aware software stack faces two main challenges: first, commercially available platforms typically do not provide means to "sense" or discover variability. Second, even if this sensing capability was available, evaluating a software stack across a statistically significant number of hardware samples and ambient conditions would prove exceedingly costly and time consuming. We introduce VarEMU, an extensible framework for the evaluation of variability-aware software that provides users with the means to emulate variations in power consumption and fault characteristics and to sense and adapt to these variations in software.

We introduce variability aware duty cycling methods and a duty cycle abstractions for embedded operating systems (TinyOS and FreeRTOS) that allow applications to to explicitly specify lifetime and minimum duty cycle or quality requirements for individual tasks, and dynamically adjust duty cycle rates and task activation schedules so that overall quality of service is maximized in the presence of power variability. We show that variability-aware duty cycling yields a 3–22x improvement in total active time over schedules based on worst-case estimations of power, with an average improvement of 6.4x across a wide variety of deployment scenarios based on collected temperature traces. Conversely, datasheet power specifications fail to meet required lifetimes by 7–15%, with an average 37 days short of a required lifetime of one year. Finally, we show that a target localization application using variability-aware duty cycle yields a 50% improvement in quality of results over one based on worst-case estimations of power consumption.

In addition to task activation control though duty cycling, a *choice* of software to be executed provides further opportunities for optimization. We introduce ViRUS (Virtual function Replacement Under Stress), an application runtime sup-

port system that adjusts service quality according to variability-aware policies. In ViRUS, different code paths implement the same function with varying quality-of-service for different energy costs. Mutations from one version to another are triggered by monitoring vectors of variability and energy stress. We demonstrate ViRUS with a framework for transparent function replacement in shared libraries and a polymorphic version of the standard C math library in Linux. Application case studies show how ViRUS can tradeoff upwards of 4% degradation in application quality for a band of upwards of 50% savings in energy consumption.

The dissertation of Lucas Francisco Wanner is approved.

Mario Gerla

Puneet Gupta

Jens Palsberg

Mani B. Srivastava, Committee Chair

University of California, Los Angeles

2014

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to extend my to gratitude the many people that helped and guided me through my journey as a Ph.D. student.

To my advisor Mani Srivastava, for ideas, support, encouragement, and constructive criticism that not only made this work possible, but also immensely enriched my whole experience at UCLA.

To Puneet Gupta, for advice and guidance that helped shape and refine much of this work.

To my committee members Jens Palsberg and Mario Gerla for their time, interest, and valuable comments.

To Rajesh Gupta, Nikil Dutt, Subhasish Mitra, and my colleagues in the NSF Variability Expedition, for early feedback on ideas and results and for forming an inquisitive community around the problem of hardware variability.

To my friends, colleagues, and collaborators in the NESL and NanoCAD labs at UCLA, especially Rahul Balani, Paul Martin, Liangzhen Lai, Salma Elmalaki, Charwak Apte, Sadaf Zahedi, Gauresh Rane, Haksoo Choi, Newton Truong, Henry Herman, Thomas Schmid, Zainul Charbiwala, and Supriyo Chakraborty.

To my mother Filomena Hoff and my sister Joice Wanner, for their constant support that travels the furthest distances.

Finally, to my wife Gabriella Simon Maia, for making me happy every day.

This material is based upon work supported in part by CAPES/Fulbright grant #1892/07-0 and by the NSF under awards # CNS-0905580 and CCF-1029030. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these agencies.

Vita

2001–2004	Undergraduate Research Assistant, Computer Science Department, Federal University of Santa Catarina, Brazil
2004	B.S. (Computer Science), Federal University of Santa Catarina
2004–2005	Teaching Assistant, Computer Science Department, Federal University of Santa Catarina, Brazil
2004–2008	Research Assistant, Software/Hardware Integration Laboratory, Federal University of Santa Catarina
2006	M.S. (Computer Science), Federal University of Santa Catarina
2008–2012	Fulbright/CAPES Scholar
2010–2011	Teaching Assistant, Henry Samueli School of Engineering and Applied Science, UCLA
2008–2014	Research Assistant, Networked and Embedded Systems Laboratory, UCLA

Publications

Lucas F. Wanner, Salma Elmalaki, Liangzhen Lai, Puneet Gupta, and Mani B. Srivastava, VarEMU: An emulation testbed for Variability-Aware Software, In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS'13)*, 2013.

Paul Martin, Lucas F. Wanner, and Mani B. Srivastava, Runtime Optimization of System Utility with Variable Hardware, Submitted to *IEEE Transactions on Embedded Computing Systems (TECS)*,2013.

Lucas F. Wanner, Charwak Apte, Rahul Balani, Puneet Gupta, Mani B. Srivastava, Hardware Variability-Aware Duty Cycling for Embedded Sensors, In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2013.

Rahul Balani, Lucas F. Wanner, and Mani B. Srivastava, Fast Iterative Optimization in Networked Cyber-Physical Systems, In *ACM Transactions in Embedded Computing Systems (TECS)*, 2013.

Yuwen Sun, Lucas F. Wanner, and Mani B Srivastava, Low-cost Estimation of Sub-system Power, In *International Green Computing Conference (IGCC'12)*, San Jose, California, June 2012.

Lucas F. Wanner, Rahul Balani, Sadaf Zahedi, Charwak Apte, Puneet Gupta, Mani B. Srivastava, Variability-Aware Duty Cycle Scheduling in Long Running Embedded Sensing Systems, In *Design, Automation and Test in Europe (DATE)*, Grenoble, France, March 2011.

Rahul Balani, Kaisen Lin, Lucas F. Wanner, Jonathan Friedman, Rajesh K. Gupta, Mani B. Srivastava, Programming Support for Distributed Optimization and Control in Cyber-Physical Systems, In *International Conference on Cyber-Physical Systems (ICCPS)* , Chicago, IL, April 2011.

Lucas F. Wanner, Charwak Apte, Rahul Balani, Puneet Gupta, Mani B. Srivastava, A Case for Opportunistic Embedded Sensing In Presence of Hardware

Power Variability, In *Workshop on Power Aware Computing and Systems (HotPower '10)*, Vancouver, Canada, October 2010.

Lucas F. Wanner and Antônio A. Fröhlich, Operating System Support for Wireless Sensor Networks, In: *Journal of Computer Science*, 4(4):272-281, 2008.

Geovani R. Wiedenhoft, Lucas F. Wanner, Giovani Gracioli and Antônio Augusto Fröhlich, Power Management in the EPOS System, In: *SIGOPS Operating Systems Review*, 42(6):71-80, 2008.

Rafael Pereira Pires, Lucas F. Wanner and Antônio Augusto Fröhlich, A Framework for Configuration and Assembly of Routing Protocols for Wireless Ad-Hoc Networks, In *International Conference and Workshop on Ambient Intelligence and Embedded Systems*, Kiel, Germany, 2008.

Rafael Pereira Pires, Lucas F. Wanner and Antônio Augusto Fröhlich, An Efficient Calibration Method for RSSI-based Location Algorithms, In: *Intl. IEEE Conference on Industrial Informatics*, Daejeon, Korea, 2008.

Lucas F. Wanner, Augusto Born de Oliveira, and Antonio Augusto Frohlich. Configurable Medium Access Control for Wireless Sensor Networks. In *International Embedded System Symposium*, pages 401–410, Irvine, CA, USA, May 2007.

Augusto B. de Oliveira, Lucas Wanner, and Antonio Augusto Frohlich. Integrating Wireless Sensor Networks and the Grid through POP-C++. In *International Embedded System Symposium*, pages 411–420, Irvine, California, US, May 2007.

Arliones Stevert Hoeller Junior, Lucas F. Wanner, and Antonio Augusto Frohlich.

A Hierarchical Approach For Power Management on Mobile Embedded Systems. In *5th IFIP Working Conference on Distributed and Parallel Embedded Systems*, pages 265–274, Braga, Portugal, October 2006.

Hugo Marcondes, Arliones Stevert Hoeller Junior, Lucas F. Wanner, and Antonio Augusto Frohlich. Operating Systems Portability: 8 bits and beyond. In *11th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 124–130, Prague, Czech Republic, 2006.

Lucas F. Wanner, Arliones Stevert Hoeller Junior, Augusto B. de Oliveira, and Antonio A. Frohlich. Operating System Support for Data Acquisition in Wireless Sensor Networks. In *11th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 582–585, Prague, Czech Republic, 2006.

Lucas F. Wanner, Arliones S. Hoeller Junior, Fauze V. Polpeta, and Antonio Augusto Frohlich. Operating System Support for Handling Heterogeneity in Wireless Sensor Networks. In *10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy, September 2005.

CHAPTER 1

Introduction

Energy management methods in embedded systems rely on knowledge of power consumption of the underlying computing platform in various modes of operation. These power specifications are usually derived from the datasheets. Unfortunately, the microelectronic substrate is increasingly plagued by variability, especially in power consumption [BKN03, ITR], both across multiple instances of a system and in time over its usage life. As a result the "datasheet power specifications" are heavily guardbanded [JKS09, GKM05] leaving much of the energy potential or application quality untapped. The major sources of variability are:

- *Semiconductor manufacturing.* Scaling of physical dimensions faster than the optical wavelengths or equipment tolerances used in the semiconductor manufacturing line has led to increased process variability [Ber06, CGK02] which makes integrated circuit designs unpredictable.
- *Environment.* Ambient condition variability (e.g., voltage in unregulated power supplies and temperature).
- *Aging.* Transistor aging (e.g., due to negative bias temperature instability [ZVR09]) can change system power/performance over time.
- *Vendor.* Multi-sourcing of parts with identical specification from different vendors is common and can cause significant variation.

Figure 1 [ITR] shows that the manufacturing variability in sleep (or static) power and total power is likely to grow over 500% and 100% respectively in the

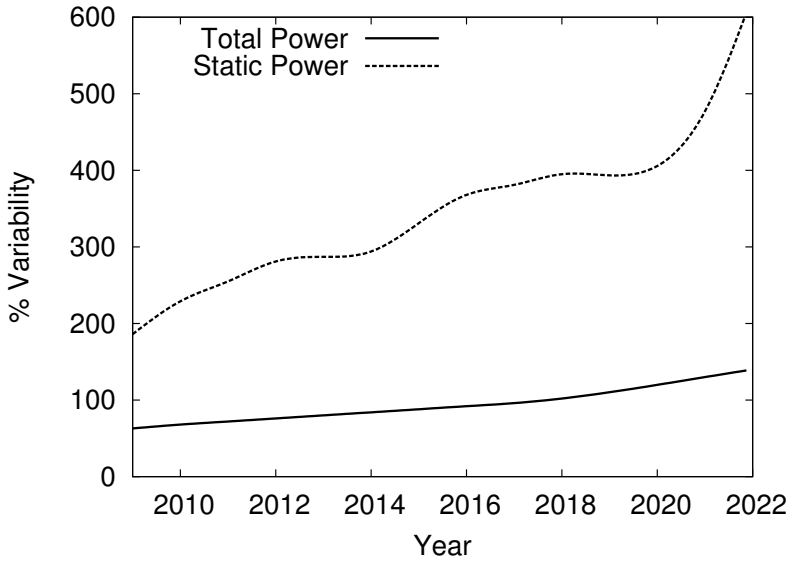


Figure 1.1: ITRS projections of power variability

next decade. Variability has been typically addressed by process, device and circuit designers with software designers remaining isolated from it by a rigid hardware-software interface, which leads to decreased chip yields and increased costs [JKS09].

Recently there have been some efforts to handle variability at higher layers of abstraction. For instance, software fault tolerance schemes have been used to address voltage [RGS09] or temperature variability [CCF07]. Hardware "signatures" are used to guide adaptation in quality-sensitive multimedia applications in [PGS11]. In embedded sensing, [MTY06, GM07] propose sensor node deployment schemes based on the variability in power across nodes.

In this work we discuss power variability for off-the-shelf embedded processors, and software approaches to handle variability in power consumption of embedded systems. In particular, we focus on instance and temperature dependent power variability that already manifests in contemporary embedded processors [BDS11]. For this, we (i) measure and characterize power for a contemporary embedded processor based on an ARM Cortex M3 core; (ii) derive projections of sleep and

active power for embedded processors with scaling of manufacturing technology; (iii) build a framework that enables testing and evaluation of variability-aware software strategies under different conditions; (iv) evaluate variability-aware duty cycle and reward-based scheduling methods under a variety of deployment scenarios and across projected future variation; (v) study the impact of duty cycle adaptation to application quality of inference; (vi) show how system-driven algorithmic choice can be leveraged to explore quality and energy cost tradeoffs in embedded applications.

1.1 Contributions

The contributions of our work are the following:

- Measurement, characterization, and projections of power variation for embedded processors. Our measurements across several instances of a contemporary ARM Cortex M3 processor show that across a temperature range of 20–60°C there is 10% variation in active power, and 14x variation in leakage power.

- Design and implementation of VarEMU, an extension to the QEMU virtual machine monitor that serves as a framework for the evaluation of variability-aware software techniques. VarEMU provides users with the means to emulate variations in power consumption and in fault characteristics and to sense and adapt to these variations in software. Through the use (and dynamic change) of parameters in a power model, users can create virtual machines that feature both static and dynamic variations in power consumption.

- Proposal, implementation, and analysis of variability-aware duty cycle adaptation methods. We show that variability-aware duty cycling yields a 3–22x improvement in total active time over schedules based on worst-case estima-

tions of power, with an average improvement of 6.4x across a wide variety of deployment scenarios based on collected temperature traces.

- Embedded operating system extensions for variability-aware duty-cycle adaptation. Two implementations, one based on the TinyOS operating system for embedded sensors and one based on the FreeRTOS real-time kernel, allow applications to express elasticity through control knobs that the operating system can tune to adjust task duty cycle, quality, and energy cost. In both implementations, a variability-aware duty cycle scheduler dynamically adjusts application control knobs so that a system-wide allowable duty cycle rate is met and overall quality-of-service is maximized.

- Analysis of the impact of duty cycle to quality of sensing. We show that a target localization application using variability-aware duty cycle yields a 50% improvement in quality of results over one based on worst-case estimations of power consumption.

- Design an implementation of ViRUS (Virtual function Replacement Under Stress), a framework for system-driven variability-aware algorithmic choice. ViRUS allows the runtime system to switch between blocks of code that perform roughly equivalent functionality at different quality-of-service levels when the system is under stress — be it in the form of scarce energy resources, temperature emergencies, or various vectors of environmental and process variability — with the ultimate goal of energy efficiency. We present a framework for function replacement in shared libraries in Linux demonstrated with a polymorphic version of the C math runtime library. We show how this system can help users in developing and analyzing tradeoffs between accuracy and energy consumption with variable hardware, and reduce the energy consumption with user-controlled quality-of-service degradation.

1.2 Related Work

Variability, thus far, has been largely addressed by process, device and circuit designers. In this context, prior work that addresses variability can be classified into (i) statistical design approaches [NS05, DBM05, KPR06], (ii) post silicon compensation and correction [GC07, KS07, TKN02], and (iii) variation avoidance [CPR04, BMR07, GBR07]. Our work differs in that it addresses hardware variability in the software layer. The closest resemblance is with [PGS11], which proposes adapting software video codec configurations based on hardware signatures. In the context of embedded sensing, our work is closest to [MTY06, GM07], which propose sensor node deployment methodologies based on variability in leakage power across different nodes.

The range of possible responses that the software can make in response to variability include: alter the computational load by adjusting task activation; use a different set of hardware resources (e.g. use instructions that avoid a faulty module or minimize use of a power hungry module); change software parameters or the hardware's operational setting (e.g., tune software-controllable control knobs such as voltage/frequency; and change the code that performs a task, either by dynamic recompilation or through algorithmic choice.

1.2.1 Task Activation Control

Variations in power consumption can be interpreted as changes in resource (energy) usage (and hence availability). Adaptation of work to resource availability is a common theme in embedded and real-time systems. In its simplest form, adaptation can be relegated to applications, with minimal system support. The system may, for example, keep track of energy consumption and notify applications to adapt in order to conserve energy when needed to meet a desired lifetime [FS99]. In energy harvesting, tasks can be adapted to cope with fluctuating

energy availability [KHZ07]. Likewise, in sensing applications, adaptive sampling mechanisms are used to minimize power consumption with different levels of data quality [AAG07].

In imprecise computation [LSL94], each task is designed to produce usable, approximate results whenever resource scarcity (e.g. due to transient failures or overloads) prevents the task from producing its desired precise result. Imprecise computation has been explored in the context of energy-aware systems, where tasks may be interrupted, producing an approximate but usable result, according to energy availability and lifetime requirements [CAC06, WWG08].

The issue of distributing available energy resources to tasks has also been explored in the literature. ECOSystem [ZEL02] introduced the concept of "Currentcy" to allocate energy resources to tasks. The system periodically distributes Currentcy to tasks, which adjust their workload according to availability. Cinder [RSL09] is an energy-aware system for mobile computing devices that features a Capacitor abstraction associated with tasks. Each capacitor represents a task's right to request energy from the system to perform its operations.

In chapter 4, we present system software strategies for variability-aware task activation control in duty cycled embedded systems. We present software stacks where applications define lifetime requirements and task activation bounds, and the variability-aware scheduler activates tasks at the maximum rate that meets that lifetime.

1.2.2 Selective use of Hardware Resources

One strategy for coping with variability is to selectively choose units to perform a task from a pool of available hardware. ERSA (Error Resilient System Architecture) [LCB10] is an architecture for probabilistic applications that consists of small number of highly reliable cores, together with a large number of cores that

are less reliable but account for most of the computation capacity. An application can be divided into control-intensive resource management code that needs to be executed on error-free hardware while data-intensive computations are often more error-tolerant.

Random access memory chips are also subject to variations in power consumption [GKGar]. Variability-aware Memory Virtualization (VaMV) defines a memory allocation architecture that allows programmers to partition their address space into regions with different power, performance, and fault-tolerance guarantees [BDN12]. Over-dimensioned register files (for example, in Graphical Processing Units used for general-purpose computation) may also be allocated in a variability-aware fashion in order to minimize errors and power consumption [RBG13].

1.2.3 Adaptation of Hardware Parameters

Variation-aware adjustment of hardware operating point, whether in context of adaptive circuits (e.g., [BKN03, GBR07, APM05]), adaptive micro architectures (e.g., [SBK06, EKD03, MJ06, TST07]) or software-assisted hardware power management (e.g., [DVA10, CLR09, TT08]) has been explored extensively in literature. A variability-aware software stack can leverage these mechanisms in an application-aware manner.

1.2.4 Adaptation of Software Parameters

Application parameters can be dynamically adapted to explore energy, quality, and performance tradeoffs [HSC11]. For example, Green [BC10] provides a software adaptation modality where the programmer provides "breakable" loops and a function to evaluate quality of service for a given number of iterations. The system uses a calibration phase to make approximation decisions based on the quality

of service requirements specified by the programmer. At runtime, the system periodically monitors quality of service and adapts the approximation decisions as needed. In the mobile context, Powerleash [Fal12] monitors energy usage of background applications in smart phones. Applications inform their rate of progress to the system through a vector of arbitrary parameters. The system learns the correlation between each vector of application progress and energy usage. When energy usage exceeds a desired budget, the system scales the progress vector and returns it to the application, which in turn adapts its work accordingly in order to meet a desired rate of battery consumption.

Adaptation of software parameters can be used to maximally leverage the underlying hardware platform in presence of variations. In the context of multimedia applications, [PGS11] demonstrated how by adapting application parameters to the post manufacturing hardware characteristics across different die, it is possible to compensate for application quality losses that might otherwise be significant in presence of process variations. In H.264 encoding, adapting parameters such as sub-pixel motion estimation, FFT transform window size, run length encoding mechanism, and size of motion estimation search window can lead to significant yield improvements, a reduction in over-design as well as application quality improvements [PGS11].

1.2.5 Dynamic Recompilation

Optimizations performed in compile-time are limited by assumptions about the target hardware for which the code is compiled. With dynamic recompilation [VE01] different optimization techniques can be tested and profiled at runtime, so that code is matched to the capabilities of the hardware which is running it. Dynamic recompilation may be performed in a system-driven manner, with minimal support from applications, providing the same adaptation knobs as in compile time optimization, e.g., loop unrolling, memory optimization, and parallelization.

1.2.6 Algorithmic Choice

One broad class of approaches for coping with hardware variability is for the software to switch to a different code path in anticipation of or in response to a variability event. Alternate code paths, or *algorithmic choice* have been explored in the energy-aware software literature. Petabricks [ACW09] and Eon [SKG07], for example, feature language extensions that allow programmers to provide alternate code paths. The runtime system dynamically chooses paths based on energy availability. In Petabricks, multiple versions of object code are created and profiled for execution time and quality using a sample set of input data. Paths for an application may be chosen statically or altered in runtime through accuracy valuations. A similar process is used in Green [BC10], where a combination of a calibration phase and runtime accuracy sampling are used by the application to define which function to execute from a set of possible candidates. In Eon, the runtime system dynamically chooses paths based on energy availability. Levels is an energy-aware programming abstraction for embedded sensors based on alternative tasks [LMM07]. Programmers define task levels, which provide identical functionality with different quality of service and energy usage characteristics. The run-time system chooses the highest task levels that will meet the required battery lifetime.

While these systems provide interesting design references for algorithmic choice, many of their assumptions do not hold true in the presence of variability. In Petabricks, execution time is the primary resource usage metric. Similarly, in Green power is assumed to be a function of execution time. With variability in active mode power, the energy cost of a fixed number of CPU cycles varies across instances and ambient conditions. Both systems rely on a calibration phase to reduce runtime overhead of evaluating quality-of-service and cost of different code paths. With variability, runtime cost of a given code path will also be variable across nominally identical devices and across the lifetime of a device, due both to

aging and changes in operating conditions. Levels triggers chances in run levels based on a history of power consumption and remaining lifetime of the system. As with the calibration phase in Petabricks and Green, a projection of future power consumption based on past history may lead to overly conservative or optimistic adaptation decisions due to variations in power consumption across time and due to ambient conditions.

While Petabricks, Eon, Green, and Levels do not consider hardware variability, similar mechanisms for expressing application elasticity can be leveraged in a variability-aware software system. For example, the application can read the current hardware signature, or register interest in receiving notifications or exceptions when a specific type or magnitude of changes occur in that signature. Application response to variability events could be structured as transitions between different run-levels, with code-blocks being activated or deactivated as a result of transitions.

Algorithms and libraries with multiple implementations can be matched to underlying hardware configuration to deliver the best performance [FJ05, LGP07]. Such libraries can be leveraged to choose the algorithm that best tolerates the predicted hardware variation and deliver a performance satisfying the quality-of-service requirement. With support from the OS, the switch to alternative algorithms may be done in an application-transparent fashion by relining a different implementation of a standard library function. In chapter 5, we present a software stack for system-driven variability-aware algorithmic choice.

1.2.7 Summary of Related Work

Table 1.1 presents a classification of selected software adaptation methods. Each method is classified according to type of adaptation and as system or application-driven. In system-driven methods, the burden of adaptation lies in the system,

Type of Adaptation	Application-Driven	System-Driven
Task Activation Control	[FS99], [KHZ07], [LSL94], [WWG08]	
Hardware Resources	[LCB10]	[BDN12]
Hardware Parameters		[DVA10], [CLR09], [TT08]
Software Parameters	[BC10], [PGS11]	[HSC11], [Fal12]
Dynamic Recompilation		[VE01]
Algorithmic Choice	[BC10], [LMM07], [ACW09], [SKG07]	

Table 1.1: Classification of Software Adaptation Methods

which provides the adaptation knobs, and initiates adaptation according some system-wide policy, potentially with input or direction from the application (e.g. bounds for task activations). In application-driven methods, the application provides the adaptation knobs, and initiates adaptation under application-specific policies, potentially with some support from the runtime system (e.g. sensors for remaining battery level).

While application-driven adaptation methods allow for wide flexibility of adaptation, they incur in increased complexity and a higher barrier of entry for application developers. Furthermore, system-wide optimization is typically not possible with application-specific methods, as individual applications can only adapt themselves and not the system as a whole. System-driven adaptation methods, on the other hand, are less flexible, as the only adaptation knobs available are those provided by the runtime system. Nevertheless, with system-driven adaptation, applications can be adapted with little or no added complexity, and optimization decisions can take the whole of the system into consideration. Our work in chapter 4 presents a software stack for system-driven task activation control where the system adapts duty cycle under application constraints. Chapter 5 presents a

variability-aware software stack for system-driven algorithmic choice.

1.3 Organization

The remainder of this text is organized as follows:

- Chapter 2 discusses power consumption variability in modern embedded processors. We measured power consumption for several instances of Atmel SAM3U processors, and found that for a temperature range of 20–60°C there is 10% variation in active power, and 14x variation in leakage power. Based on these measurements, we model sleep and active power as a function of instance and temperature, and project power consumption for advanced manufacturing technologies.

- Chapter 3 discusses evaluation strategies for variability-aware software and introduces the VarEMU emulator and its supporting software stack.

- Chapter 4 discusses variability-aware duty cycling. We explored *variability-aware duty cycle schedulers* for TinyOS and FreeRTOS where application modules specify to a range of acceptable activation parameters, and the scheduler selects the actual duty cycle based on run-time monitoring of instance-dependent power-temperature models. In an evaluation with ten instances of Atmel SAM3U processors, we found that variability-aware duty cycling yields a 3–22x improvement in total active time over schedules based on worst-case estimations of power, with an average improvement of 6.4x across a wide variety of deployment scenarios based on collected temperature traces. Conversely, datasheet power specifications fail to meet required lifetimes by 7–15%, with an average 37 days short of a required lifetime of one year. Finally, a target localization application using variability-aware duty cycle yields a 50% improvement in quality of results over one based on

worst-case estimations of power consumption.

- Chapter 5 presents a framework for system driven algorithmic choice for variability aware software featuring function replacement in shared libraries in Linux and demonstrated with a polymorphic version of the C math run-time library. We show how this system can help users in developing and analyzing tradeoffs between accuracy and energy consumption with variable hardware, and reduce the energy consumption with user-controlled quality-of-service degradation.

- Chapter 6 presents our concluding remarks.

CHAPTER 2

Power Consumption Variability in Embedded Processors

Power consumed in an embedded class microprocessor chip is broadly classified into active mode and sleep mode. In this chapter we study the temperature dependence of active and sleep mode power consumption in embedded processors and propose models to characterize it.

Our measurements show sleep and active power as a function of temperature across several instances of Atmel SAM3U microcontrollers in LQFP144 packages. While we have been unable to determine from available literature the precise technology node the chip is fabricated in, indirect evidence as well as the vintage suggests that it is most likely fabricated in a 130 nm process. The class of low-end 32-bit processors represented by the Cortex M3 is suitable for embedded applications where nodes perform data collection, aggregation, and inferences in a duty-cycled fashion. The variations we observed with the SAM3U are comparable to those found in other similar embedded processors [BDS11].

2.1 Experimental Setup

For our measurements, we used ten identical SAM3U-EK development boards. These boards feature jumpers that allow power measurements for different components. We measured current and voltage on going into the SAM3U core, with all peripherals except for the real time clock disabled. To obtain synchronized

voltage and current measurements we used a pair of Agilent 34410A digital multimeters with a basic accuracy of 0.06%, externally triggered by a function generator. Each measurement point represents the average power dissipated by the core across 50,000 measurements, with a sampling rate of 1,000 samples per second. We used a TestEquity 115F temperature chamber allowing control of ambient temperature with $\pm 0.5°C$ accuracy. Figure 2.1 illustrates our test setup.

2.2 Sleep Mode Power Consumption

With shrinking geometries the ratio of sleep mode power to active mode power has been increasing (as high as 40% in chips fabricated using 65nm technology) [PSB05]. This is due to the inability to turn devices "off" effectively as device dimensions continue to shrink. Manufacturing spread in transistor parameters can cause up to 20x variation in sleep mode power [BKN03] in addition to substantial variation with supply voltage and temperature. Specifically in context of embedded sensor platforms, which often are deployed in extreme ambient conditions, the variation in leakage power during the lifetime of a device may be substantial.

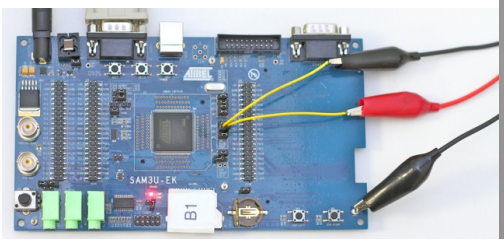
2.2.1 Analytical Modeling of Sleep Power

Sleep power has four main sources:

1. sub-threshold leakage current that flows between source and drain of a MOSFET for gate-to-source voltages below the threshold,

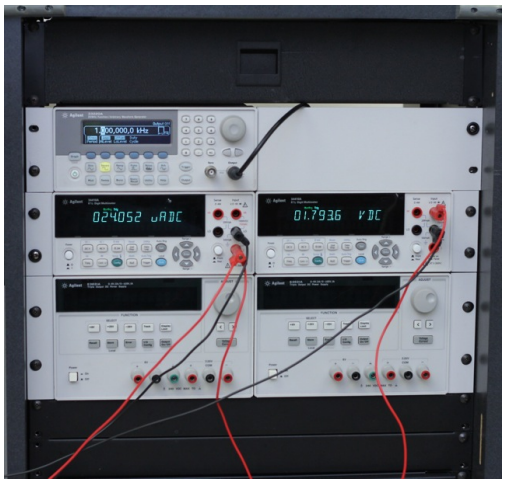
2. gate leakage current due to tunneling of carriers through the gate oxide to the substrate,

3. reverse-biased junction leakage current which flows from the source/drain regions to the substrate through the reverse biased p-n junctions due to



(a) SAM3U-EK Board

(b) Temperature Chamber



(c) Power Measurement Setup

Figure 2.1: Experimental setup

band-to-band tunneling and diffusion, and

4. gate-induced drain leakage current due to band to band tunneling in the region of overlap between the gate and drain.

At temperatures below 150°C, only the first two components are large enough, and of the two only sub-threshold leakage exhibits strong variability with temperature. Therefore, sleep power can be modeled as the following function of temperature (derived from BSIM4 compact device model [BSI]):

$$P_{sleep} = V_{dd}(AT^2 e^{-B/T} + I_{gl}) \tag{2.1}$$

where A and B are technology-dependent constants, I_{gl} is the temperature-independent gate leakage current, and T is the core temperature. Coefficients in the model are fitted to individual instances, and hence capture both temperature and instance-dependent variability. We combine the sleep power model with a model of the thermal dynamics of a packaged chip [HQF10]:

$$RC\frac{dT(t)}{dt} + T(t) - RP(t) = T_{amb} \tag{2.2}$$

where $T(t)$ and $P(t)$ are the core temperature and power consumption of the chip at time t, R and C are the thermal resistance and capacitance of the chip package, and T_{amb} is the ambient temperature. At steady state $\frac{dT(t)}{dt} = 0$, so that $T_{steady-state} = T_{amb} + RP(t)$.

For the SAM3U in a LQFP144 package, the typical values of R and C are 50°C/W and 4–5 J/°C respectively. The nominal static power of SAM3U is 30 μW. Nominal active power when operating at 4 MHz while performing a Dhrystone benchmark is 9 mW. From (2.2), when sleep mode power measurements are performed, the self-heating of the chip due to static power consumption is

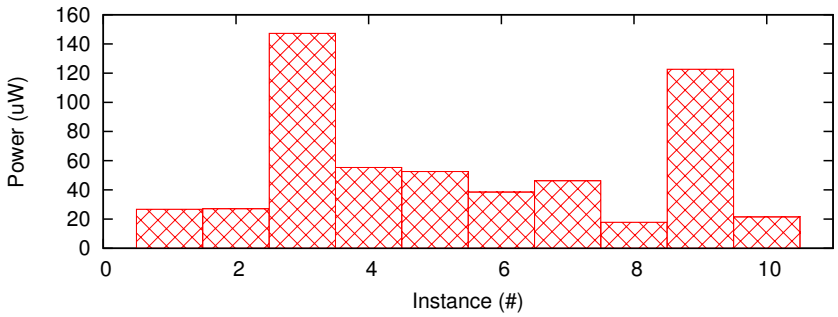


Figure 2.2: Sleep power at room temperature

negligible, and in active mode the temperature difference between ambient and core temperature is $\sim$0.5°C.

2.2.2 Experimental Measurements

Based on the preceding analysis, it is reasonable to assume that the static power follows a similar dependence on ambient temperature as given by (2.1). We verify this assumption through measurements and characterize each instance of microcontroller based on the above model.

For leakage measurements, we disable all peripheral devices in the SAM3U except for the Real-Time Clock (RTC), select the chip's internal 32kHz RC oscillator as clock source, configure the chip for wakeup with an RTC interrupt, and execute the "wait for event" instruction, which causes the processor to enter *sleep* mode. In addition to *sleep*, the SAM3U microcontroller features two other low power modes. The first, *backup*, completely powers off the core. While this results in the lowest possible power consumption, it also results in wakeup times more than 50 times larger than in other modes, and therefore is not practical for duty-cycled systems. The second low-power mode, *wait*, allows fast wakeup with some specific clock configurations and wakeup sources. In our measurements, we found power dissipated in *wait* mode to be equivalent to power in *sleep* mode with the aforementioned configuration.

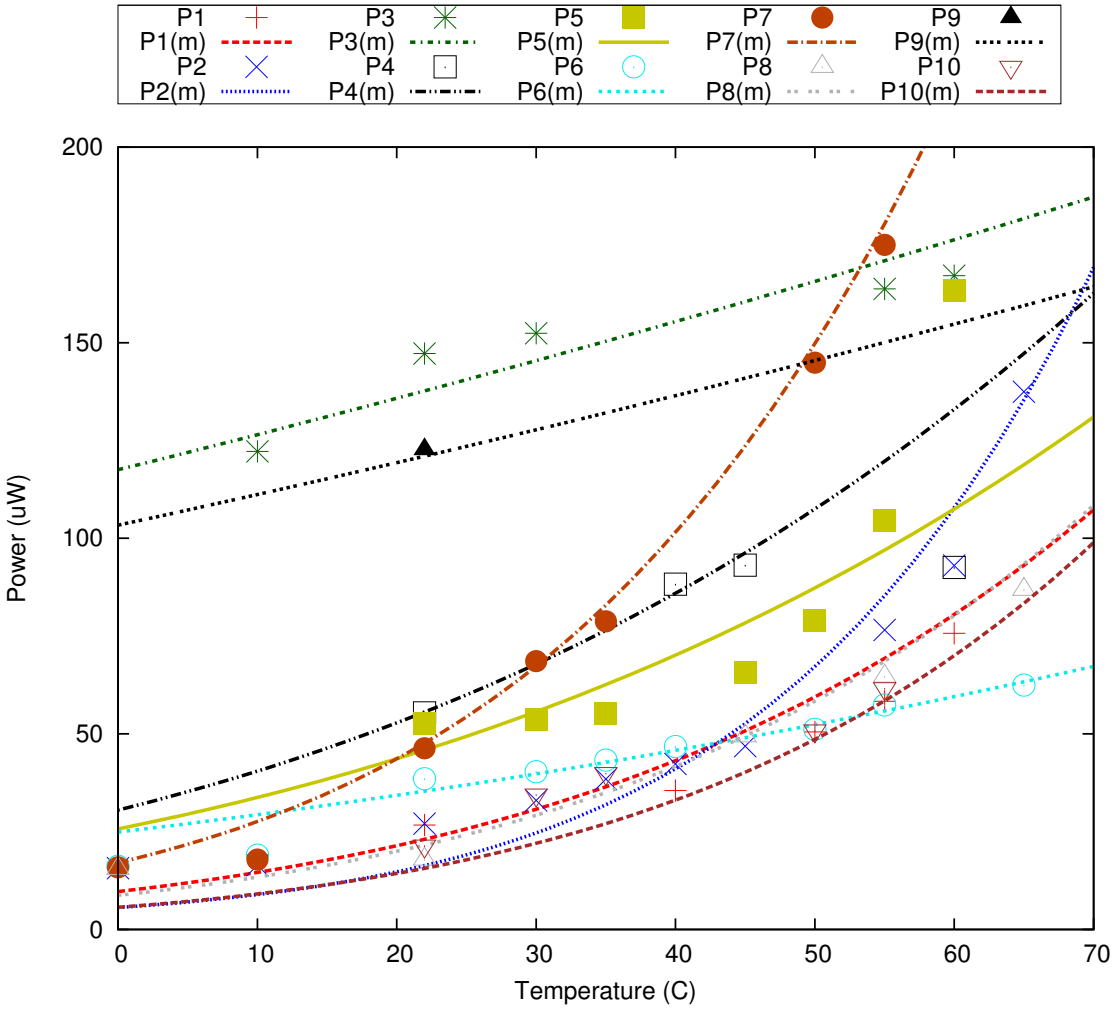


Figure 2.3: Measured and modeled variability of sleep power with temperature

Figure 2.2 shows that the variation in sleep power across ten instances of SAM3U at room temperature is approximately $8x$. Figure 2.3 shows the experimental data for sleep power consumption of the SAM3U instances across a temperature range fitted to the analytic model discussed earlier, using minimum mean square error criterion. As expected, individual processor instances exhibit large sleep power variations over the temperature range. While change in sleep power for any individual processor is monotonic, the magnitudes of variations are different so that relative rankings of different processors change over temperature. Root mean square error between measurements and model across all instances is 6.7 μW. Over a temperature range of $20 - 60$°C, which is representative of the temperatures that embedded sensors deployed under unregulated and extreme ambient conditions often face (e.g. in factories, desert, etc.), total variation across all ten instances is 14x.

2.3 Active Mode Power Consumption

Active power has three main components: switching power, short circuit power, and active mode leakage. Switching power is consumed when devices switch between different logic values, and is given by $P_{switching} = \alpha C V_{dd}^2 f$, where α is the activity factor that represents how often a gate switches per clock cycle, C is the capacitance at the switching node, V_{dd} is the supply voltage and f is the clock frequency. Short circuit power is dissipated during the time period when inputs ramp between logic levels and there is a direct path between supply to ground. Finally, devices that do not switch while in active mode also consume power. Leakage power has an exponential dependence on temperature, as elaborated in section 2.2.1. For our experiments, however, P_{active}/P_{sleep} is nominally $\sim$300. Hence this component has a small effect only on the high temperature range dependence of the total active mode power.

2.3.1 Analytical Modeling of Active Power

The temperature dependence of active mode power can be explained by dividing the temperature range of interest into three regimes [WBA11].

(i) Low temperature regime: The contribution of active mode leakage is negligible as at low temperatures. An increase in threshold voltage V_{th} causes exponential decrease in leakage. The capacitance increase is a linear function of temperature and hence contributes to the linear dependence of active mode power. At low temperatures, inverse temperature dependence causes the decrease in active mode power to have a steeper slope as temperature reduces. The relationship with temperature is characterized as:

$$P_d = P_{T_1} + k_1(T - T_1)^{\alpha} \tag{2.3}$$

where, $0 < \alpha < 1$, $T_1 < T < T_2$, k_1 and α are fitting parameters. $T_1 < T < T_2$ defines the low temperature regime.

(ii) Nominal temperature regime: Short circuit and switching power have a linear dependence in this regime, and the active mode leakage is negligible. The relationship with temperature is characterized as:

$$P_d = P_{T_2} + k_2(T - T_2) \tag{2.4}$$

where, $T_2 < T < T_3$ and k_2 is a fitting parameter. $T_2 < T < T_3$ defines the nominal temperature regime.

(iii) High temperature regime: The switching power varies linearly (capacitance dependence is linear), short circuit power and active mode leakage give active mode power in this regime a super-linear dependence [WAB13]. The relationship with temperature is characterized as:

$$P_d = P_{T_3} + k_3(T - T_3)^{\beta} \tag{2.5}$$

where, $1 < \beta$, $T_3 < T$, k_3 and β are fitting parameters. $T_3 < T$ defines the high temperature regime.

2.3.2 Experimental Measurements

Figure 2.2 shows that the variation in active power across ten instances of SAM3U at room temperature is approximately 4%. Figure 2.5 shows the active power model fitted to our measured data for active mode power consumption across a temperature range. For this experiment, we used the same measurement setup as in the sleep mode power measurements. All SAM3U peripherals were disabled, except for the RTC. The core was clocked from the internal ring oscillator at 4MHz, continuously running a Dhrystone benchmark program. Root mean square error between measurements and model across all instances is 0.02 mW. Over a range of 20–60°C, total variation across all ten instances is 10%.

Interestingly, for the SAM3U processors we used, even the clock frequency varied by 2% across the temperature range and 6% across different instances as shown in Figure 2.6. To measure clock frequency, we toggled an I/O pin at a rate proportional to the core frequency, and observed the resulting frequency at different temperatures with a digital oscilloscope. All processors were set to operate nominally at 4 MHz. The clock is generated by an internal ring oscillator (RO). Generally, RO frequency decreases with an increase in temperature so there is always temperature compensation provided with the RO circuit to generate the clock at the specified frequency across temperature. We observe that the frequency increases with temperature. This is likely due to temperature over-compensation.

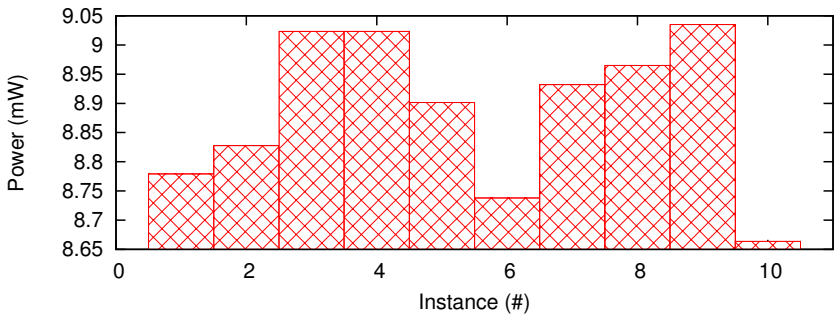


Figure 2.4: Active power at room temperature

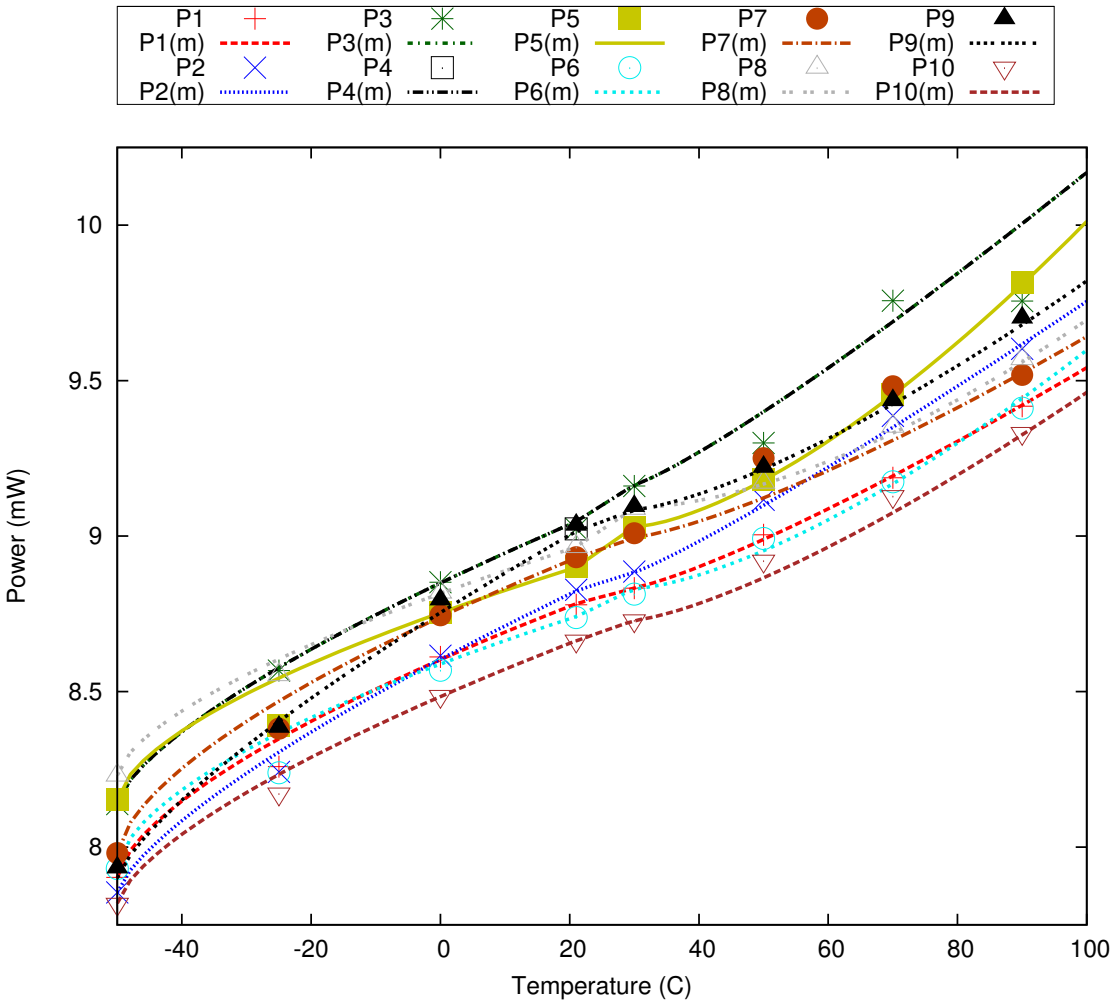


Figure 2.5: Measured and modeled variability of active power with temperature

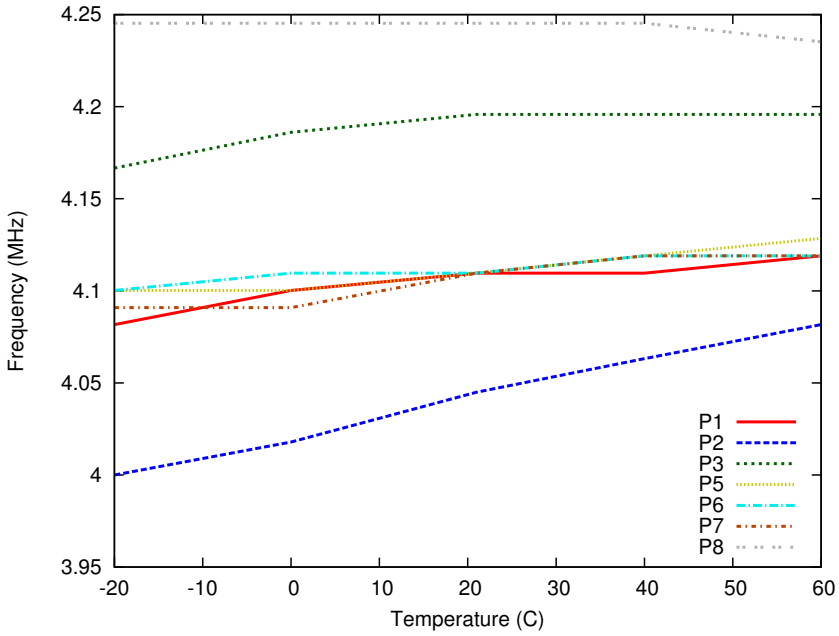


Figure 2.6: Measured frequency variation with temperature

2.4 Power Projections for Advanced Technologies

The characterization discussed above was based on actual measurements from hardware manufactured in 130nm. Nominal power consumption variability is characterized in ITRS across sub-130 nm technology nodes. We use this data to project the spread of power vs. temperature curves. PTM 130, 90, 65 and 45 nm spice models were used to project the scaling of active and sleep mode power across these technology nodes based on a ring oscillator design. Relative temperature dependence was assumed to be the same as observed in the measurements. We assume that the temperature regimes remain constant across technologies. The projections can be refined by calibrating these regimes for advanced technology using real measurements. We also assume in our experiments that while in active mode 20% of devices are switching. Tables 2.1 and 2.2 list typical sleep and active mode power model parameters at each technology node. Figures 2.7 and 2.8 show expected projection of nominal sleep and active power across temperature resulting from these parameters for each node technology.

Instance	A	B	I_{gl}	V_{dd}
Typ_130	1.0	2605.5	0.0	1.8
Typ_90	1.26	2400	0.2	1.8
Typ_65	1.76	2300	0.6	1.8
Typ_45	2.52	2100	1	1.8

Table 2.1: Sleep power model parameters across technologies

Instance	k_1	k_2	k_3	a	b
Typ_130	0.0511	0.0095	0.003325	0.669	1.3456
Typ_90	0.0246	0.0046	0.0016	0.71	1.5228
Typ_65	0.0094	0.00175	0.000615	0.735	1.6335
Typ_45	0.0046	0.000855	0.0003	0.755	1.722
	$P_{T1}(mW)$	$P_{T2}(mW)$	$P_{T3}(mW)$	T_2 (°C)	T_3 (°C)
Typ_130	7.7	8.6	8.68	21	30
Typ_90	2.82	3.33	3.37	21	30
Typ_65	0.9	1.125	1.14	21	30
Typ_45	0.364	0.479	0.4867	21	30

Table 2.2: Active mode power model parameters across technologies

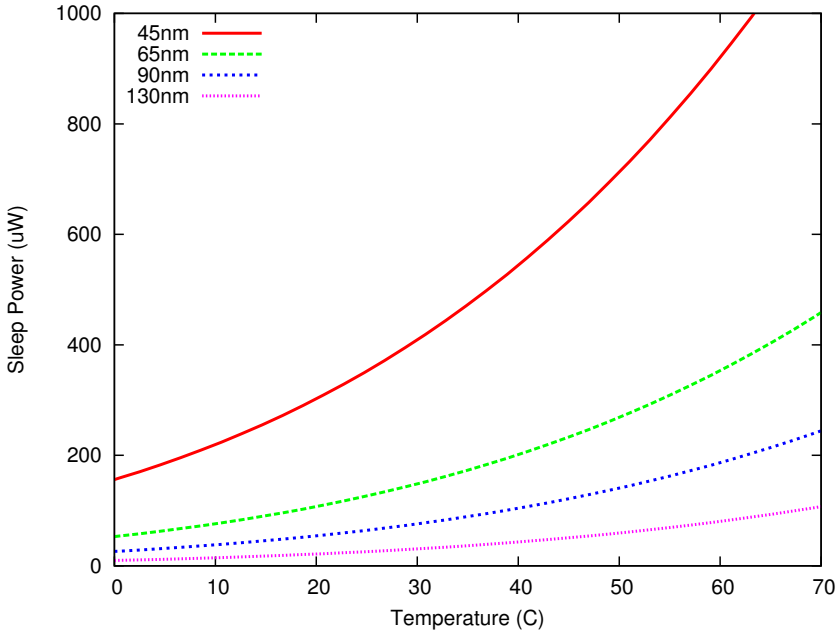


Figure 2.7: Sleep power projection across technologies

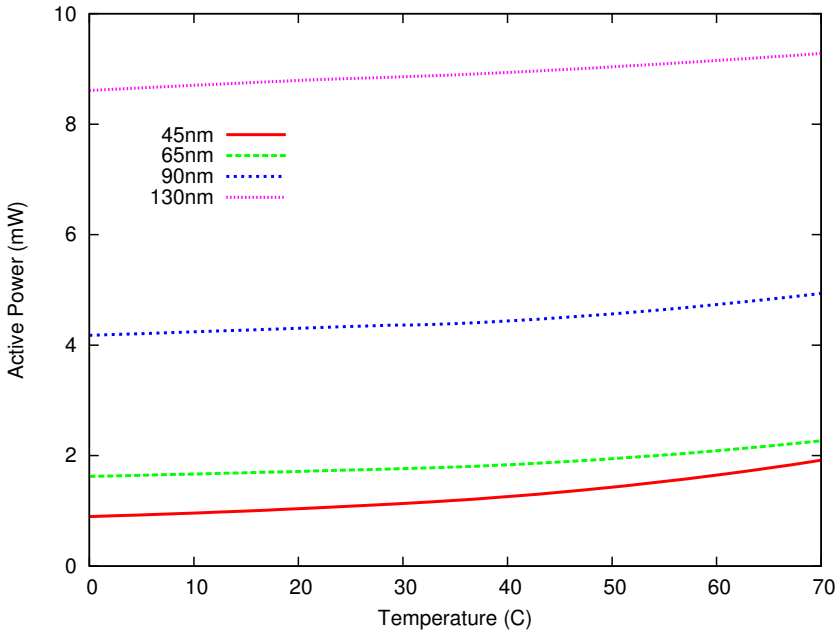


Figure 2.8: Active power projection across technologies

CHAPTER 3

Evaluation of Variability-Aware Software

The evaluation of a variability-aware software stack faces two main challenges: first, commercially available platforms typically do not provide means to "sense" or discover variability. Second, even if this sensing capability was available, evaluating a software stack across a statistically significant number of hardware samples and ambient conditions would prove exceedingly costly and time consuming.

In hardware design, simulations at various levels of abstraction can be used to evaluate the impacts of hardware variability due to PVT (Process, Voltage, and Temperature) variations and circuit aging. While gate- and RTL-level simulators can co-simulate both software and hardware, their runtimes are orders of magnitude slower than real-time [CDB09]. Cycle-accurate architecture-level simulators like Wattch [BTM00], with typical runtimes of 2-3 orders of magnitude slower than real-time, suffer from the same problem. FPGA-based emulators like [KVR11, CLM12] can achieve similar runtime as real-time, but offer limited observability and controllability, and suffer from poor portability and flexibility.

We introduce VarEMU, an extensible framework for the evaluation of variability-aware software. VarEMU provides users with the means to emulate variations in power consumption and fault characteristics and to sense and adapt to these variations in software. VarEMU is an extension to the QEMU virtual machine monitor [QEM13], which relies on dynamic binary translation and supports a variety of target architectures with very good performance. For many target machines, QEMU provides faster than real time emulation. Because QEMU can run

unmodified binary images of physical machines, VarEMU enables the evaluation of complete software stacks, with operating system, drivers, and applications.

In VarEMU, timing and cycle count information is extracted from the code being emulated. This information is fed into a variability model, which takes configurable parameters to determine energy consumption and fault variations in the virtual machine. Energy consumption and susceptibility to faults are also subject to dynamic change according to an aging model. Control over faults and virtual energy sensors are exported as "variability registers" mapped into memory that is accessible to the software being emulated, closing the loop. This information is exposed through a variability driver in the operating system, which can be used to support software adaptation policies. Through the use of different variability emulation parameters that capture instance-to-instance, environmental, and age-related variation, VarEMU allows users to evaluate variability-aware software adaptation strategies across a statistically significant number of hardware samples and scenarios.

3.1 VarEMU Architecture and Implementation

Figure 3.1 presents an overview of the VarEMU architecture. Applications in a virtual machine interact with VarEMU by querying for energy, cycle count, and execution registers for different classes of instructions and by allowing or disallowing faults in the execution of emulated instructions. An operating system driver mediates the interaction of applications with a virtual hardware device which exposes the VarEMU interface to the VM. On VMs without operating systems, applications handle this interaction directly.

When starting VarEMU, users provide a configuration file that sorts instructions into different classes and parameters to a model that is used to determine power consumption for each of the classes. These parameters are subject to dy-

namic change during runtime according to an aging model. Users may change parameters for the power model dynamically (e.g. to emulate variations in power consumption due to changes in temperature, the user would periodically change the temperature parameter of the power model). Users may also query the VM's cycle counters and energy registers.

Whenever an instruction is executed in the virtual machine, the cycle counter for its instruction class is incremented. Energy expenditure for a class of instruction is determined as a function of accumulated execution time for all instructions in that class and power consumption for the class as determined by a power model.

For instructions configured by the user as *susceptible to faults*, the execution of translated code may be preceded, succeed, or replaced with alternative, *faulty* operations. These operations may, in turn, cause changes to cycle counting (e.g. due to a less precise version of the instruction taking fewer cycles to complete) or change parameters in the power model (e.g. voltage or frequency). Faults are injected only when explicitly activated by emulated software. A runtime parameter passed from emulated software to the fault module when enabling faults allows users to configure which faults are enabled and/or the nature of faults (e.g. precision of a numerical operation). This allows users to study the effects of faults in instruction execution on individual applications phases, without compromising the stability of the runtime system. The remainder of this section describes the architecture and implementation of VarEMU.

3.1.1 Cycle and Time Accounting

We account time in VarEMU on an *instruction class* basis. Each instruction is associated with a user-defined class. A data structure holds total number of cycles and time spent executing instructions of each class. To associate instructions with classes, each instruction in a translation block is augmented with an informa-

tion structure (`vemu_instr_info`) containing fields for the instruction operation code (opcode), instruction name, instruction class, number of cycles, fault status, and the instruction word itself.

When a new instruction word is found, its opcode is decoded, and the instruction information structure is filled with its corresponding default values. An input file in JSON format allows users to change the default number of cycles, class, and fault status for any instruction. The number of cycles may also be altered by the fault module at runtime.

A *helper* function in QEMU allows calling arbitrary functions from translated code. We use one such helper to perform a call to a function that increments the number of cycles for a given instruction class after each instruction is executed (`vemu_increment_cycles`). This function adds the number of cycles in the instruction's information structure to the total number of cycles for its instruction class. Likewise, it increments total active time for that instruction class, based on current (virtual) frequency. In processors where the number of cycles taken by an instruction is not constant, information from the instruction word (e.g. input registers used, immediate values) could be used to accurately determine the number of cycles.

We must also account for cycles spent in standby or *sleep* modes. In many architectures, a special instruction (e.g. `WFI` in ARM, or `HLT` in x86 processors) puts the processor in standby mode. After this instruction is issued, the processor will not execute other instructions until an interrupt (typically from a timer or an external device) is fired. Keeping track of real sleep time (i.e., reflecting hardware timing) is important for applications (e.g., in energy-aware duty cycling), as well as for circuit aging models. When we encounter such an instruction, we store a timestamp with current VM time. When an interrupt occurs following standby we read a new timestamp, and add the time difference to the counter for total time spent in sleep mode.

Because QEMU runs virtual machines as *best-effort*, the actual execution frequency of emulated instructions may not match the (virtual) frequency of the hardware. If the VM never enters standby mode, there will be no adverse effects other than a discrepancy between total virtual time accounted with the cycle counters and wall clock time elapsed. If the VM does enter a standby mode, time spent in that mode must be adjusted to reflect hardware behavior.

Consider, for example, a system with periodic tasks where processor utilization is less than 100%. After the system completes tasks, it goes into standby mode, and waits for a timer interrupt corresponding to the next period. Figure 3.2a illustrates such a system, where processor frequency is 100 MHz, timer frequency is 1 Hz, task execution takes 50 M cycles (0.5 seconds), and time spent in standby mode is 0.5 seconds. If emulated execution is faster than hardware, sleep time in the VM would be greater than in hardware. Conversely, if emulation is slower than hardware, sleep time in the VM would be smaller than in hardware.

In order for sleep time accounting in VarEMU to reflect hardware timing, we keep track of emulated execution time for each active time cycle. When a sleep cycle is initiated, we calculate the delta between virtual execution time (from our cycle counters, reflecting hardware execution time) and emulated execution time for the last active period. We then deduct this delta from the sleep time interval. Figure 3.2b illustrates our solution. In cases where processor utilization in hardware is 100%, but emulated execution time is faster than hardware, it is possible for the sleep time interval to be negative. In this case, the hardware version of the processor would continue executing immediately after the standby instruction. We emulate this by returning a sleep interval of 0. The converse situation (emulated time is slower than virtual time) does not lead to a problem, as after continuing execution immediately after the standby instruction we deduct a negative delta from an interval of zero, leading to the correct sleep time interval.

3.1.2 Energy Accounting

Energy consumed by an instruction of a given class is determined as a function of execution time (number of cycles divided by frequency) and power for that class. Power is in turn determined by a model with arbitrary parameters (minimally, voltage and frequency). By fitting the power model with different parameters, users can emulate instance-to-instance variation. By changing parameters dynamically, users can emulate the effects of dynamic or environmental variation (e.g. due to changes in supply voltage or temperature). Power model parameters may also be dynamically changed with an aging model.

While active and sleep time are accounted on a per-event basis (i.e. on each instruction or sleep cycle), energy is accounted on demand, i.e. only when a read command is issued from emulated software or external monitor, or when one of the power model parameters change. For each energy accounting event, we keep track of sleep time and active time for each class of instructions since the last event, and accumulate energy for each interval in the appropriate energy registers. There is one active energy register per instruction class, and for sleep energy.

Energy accounting is independent of power model, so that users may define their own models. A power model implements three functions: The first function returns active power in Watts for a given class of instruction. The second returns sleep power in Watts as a function of standby mode (e.g. clock gated, power gated). The final function is used to change power model parameter n of class c to value v. Any power model must also define at least two parameters: frequency and voltage. The default power model for VarEMU, presented in Section 3.1.4 defines several additional parameters to capture static and dynamic variability.

3.1.3 NBTI Aging Model

Negative bias temperature instability (NBTI) is a circuit wear-out mechanism that will degrade the PMOS threshold voltage (V_{thp}) and thus the circuit performance. To model the NBTI-induced aging effect in VarEMU, we use the analytical model for the $|V_{thp}|$ degradation of a MOS transistor as in [CWC12, BWV06, WYB07].

$$
\begin{aligned}
|\Delta V_{thp}| &= \left(\frac{\sqrt{K_v^2 T_{clk} \omega}}{1 - \beta_t^{1/2n}} \right)^{2n} \\
\beta_t &= 1 - \frac{b_1 + \sqrt{b_2 (1-\omega) T_{clk} exp(b_5/T)}}{b_3 + b_4 \sqrt{t}} \\
K_v &= b_4 (V_{dd} - V_{thp}) exp(b_5/T)
\end{aligned}
\tag{3.1}
$$

where V_{dd} is the supply voltage, b_1, b_2, b_3, b_4, b_5 are technology-dependent parameters. T_{clk} is the time period of one stress-recovery cycle, ω is the duty cycle (the ratio of the time spent in stress to time period), t is the total lifetime of a transistor, n is a time exponent equal to 1/6 for an H_2 diffusion model. Since NBTI-induced degradation is insensitive to the switching frequency when it is larger than 100Hz [BWV06], similar to [WYB07], we assume $T_{clk} = 0.01s$ in this work.

Based on the aging model in (3.1), the key activity-related parameters are the duty cycle ω and total lifetime t. In VarEMU, we use the cycle counting feature to implement the bookkeeping function for activity-related parameters, i.e. total normal runtime t_n and total runtime under power gating t_{pg}.

Since NBTI-induced degradation depends on the exact signal switching pattern, VarEMU reports the upper and lower bound aging scenarios. The upper bound of the aging scenario will be $t = t_n + t_{pg}$ and $\omega = t_n/t$. The lower bound of the aging scenario will be $t = t_n + t_{pg}$ and $\omega = 0.5t_n/t$. Since the model in (3.1) assumes a periodic stress-recovery pattern, this model may not be adequate to accurately capture NBTI effects under some dynamical scenarios like dynamic voltage scaling and long-term power-gating. Enabling dynamic will require either

more sophiscated aging models or aging simulators as in [CSG11] (too slow for our purpose).

3.1.4 Aging-aware Power and Delay Model

In this section we present the default power model for VarEMU which accounts for aging effects. The processor power consumption can be classified as active power and sleep power. Active power includes switching power and short circuit power. In VarEMU, we use the switching power model as in [RCN96]:

$$P_{switching} = \sum_{i=1}^{n} C_i \beta_i V_{dd}^2 f \tag{3.2}$$

where C_i is the equivalent switching capacitance for each instruction class i, β_i is the fraction of class i instructions in all instructions, and f is the clock frequency.

We use the short circuit power model as in [Vee84]:

$$P_{short} = \sum_{i=1}^{n} \eta_i (V_{dd} - V_{thn} - V_{thp})^3 f \tag{3.3}$$

where η_i is a technology- and design-dependent parameter for instruction class i, V_{thn} is the threshold voltage for NMOS, and V_{thp} is the threshold voltage for PMOS and equals $|V_{thp0} + \Delta V_{thp}|$, V_{thp0} is the threshold voltage without degradation.

The sleep power can be modeled as:

$$P_{sleep} = V_{dd}(I_{sub} + I_g) \tag{3.4}$$

where I_{sub} is the subthrshold leakage current and I_g is the gate leakage current.

The leakage current models can be derived from the device model in [BSI]. We simplify the model and extract the temperature- and voltage-dependency as:

$$I_{sub} = a_1 T^2 (exp(\frac{-a_2 V_{thp}}{T}) + exp(\frac{-a_2 V_{thn}}{T}))exp(\frac{-a_3 V_{dd}}{T}) \tag{3.5}$$

where a_1, a_2, a_3 are empirical fitted parameters.

We use the gate leakage model from [KAB03]:

$$I_g = a_4 V_{dd}^2 exp(-a_5/V_{dd}) \tag{3.6}$$

where a_4, a_5 are empirical fitted parameters. There are secondary effects of temperature on some parameters such as threshold votage and electron mobility, but the effects are neglagible for our purpose.

The dependence of circuit delay d on supply voltage V_{dd} and threshold voltage can be modeled by the alpha-power law [SN90]. Since NBTI has effect only on PMOS (PBTI on NMOS respectively), due to the complementary property of CMOS, the overall circuit delay can be modeled as:

$$d = \frac{K_p C_p V_{dd}}{(V_{dd} - V_{thp})^\alpha} + \frac{K_n C_n V_{dd}}{(V_{dd} - V_{thn})^\alpha} \tag{3.7}$$

where C_p and C_n are equivalent load capacitances for PMOS and NMOS respectively, K_p, K_n and α $(1 < \alpha < 2))$ are technology and design dependent constants.

In this work, we use a commercial 45nm process technology and libraries as our baseline. The aging model is fitted to the NBTI aging equation given in the technology design manual. The power and delay model parameters are fitted to the SPICE simulation results of a inverter chain using device model given in the technology libraries. Compared to the power and delay value reported by SPICE results, errors in our model are less than 2% for $0.8V < V_{dd} < 1V$, $0mV < |\Delta V_{thp}| < 50mV$ and $10°\text{C} < T < 90°\text{C}$.

Although the absolute power and delay values of the entire processor may not match the results of the inverter chain, we expect their sensitivity to voltage and temperature to follow similar trends if the inverter chain is designed to match the same design properties (e.g., cell types, fan-out ratio) of a particular processor design. In this work, the final power and delay values are normalized to the measured data obtained from a Cortex M3 testchip using the same technology.

3.1.5 Faults

VarEMU allows faults to be inserted before or after, or to completely replace the execution of an instruction. A *faulty* implementation of an instruction in VarEMU is an arbitrary C function that has access to the complete architectural state of the VM, and hence may manipulate memory, general purpose registers, and status and control registers. Faulty versions of instructions may co-exist with its respective correct versions, and faults may be dynamically enabled and disabled from emulated software.

When an instruction is disassembled, we check its VarEMU field to determine if it is susceptible to faults. For instructions with `pre` and `post` execution faults, we simply generate code that calls the respective fault helper functions at execution time. These helper functions determine whether the fault will occur, and conditionally call the fault implementation. For instructions with `replace` faults, the code generation process is more complex: if we simply called a `replace` helper, the developer of the replacement fault would also have to implement a correct version of the instruction. Hence, we generate two code paths, one for the faulty path, and one for the original instruction (for when faults do not occur). The faulty path is always called, and returns a boolean value which determines whether the original instruction should be executed or not. This is accomplished with the equivalent of a conditional branch instruction, which jumps to the end of the current translation block if the return value of the replace helper is not zero.

All of the following conditions must be met in order for a fault to occur: 1) the instruction under execution is marked as subject to faults; 2) the processor is not in a privileged mode (e.g., faults are not permitted in the OS kernel); 3) faults have been enabled by emulated software; 4) user-defined conditions, e.g., based on conditional or random variables. If these conditions are not met, the original version of the instruction will be executed without faults.

Figure 3.3 shows a simple example of a *stuck-at* fault in the multiply instruction. If the processor is currently running in privileged mode, or if faults have not been enabled from emulated software, the function returns zero, which causes the original instruction to be executed. Otherwise, the instruction operation code is decoded. For the multiply opcode, the source and target registers are decoded, and the multiply operation is augmented with the stuck-at-one fault. The result is written into the destination register.

While the fault presented in Figure 3.3 is deterministic in nature (a stuck-at-one in the LSB of the target register) and occurrence (always happens when faults are enabled in non-privileged mode), users may include additional implementations or conditions for faults, e.g., based on history, random variables, architectural state, or operational parameters such as voltage and frequency in the power model. Users may also call external software modules (e.g. RTL simulators) from the fault module in order to model realistic faults that, for example, take spacial correlation or instruction inter-dependency into account. Faulty execution may in turn influence cycle counting (e.g. a faulty version of an instruction that finishes in fewer cycles) or energy accounting (e.g. a faulty version of the instruction that is less power intensive).

3.2 Software Interfaces

VarEMU allows users and external software to configure instruction information (class of instruction, susceptibility to faults), dynamically change power model parameters, and query the VM for cycle, time, and energy information.

An input file in JSON format specifies instruction classes and power model parameters for a VM. A class of instructions is defined by an index, a name and a list of instruction names. By default, all instructions are linked to a single *catch-all* class. Instructions not listed in the input file remain linked to the default class. A

dictionary links each instruction class with its respective list of power model parameters. A minimal input file includes only a list of power model parameters for the *catch-all* instruction class. The input file may also define lists of instructions susceptible to each type of fault supported by VarEMU.

QEMU provides a monitor architecture for external interaction with the VM. This monitor listens for commands and sends replies on an I/O device (e.g. `stdio` or a socket). We extended this monitor to provide commands to query a VM's energy, cycle, and time information, and to dynamically change power model parameters. Inputs and responses to and from the monitor are in JSON format. A `query-energy` command returns accumulated energy for sleep mode and for each instruction class. Similarly, a `query-time` command returns accumulated execution and sleep times. Finally, a `change-model-param` command allows users to change power model parameter n of class c to value v.

A combination of the `change-model-param` command described above and the standard `stop` and `cont` commands provided by QEMU allows users to systematically emulate dynamic variations in power consumption due to environmental factors (e.g. changes in ambient temperature).

We implemented a small application that demonstrates interaction with the VarEMU monitor commands. This application queries the monitor every second for energy and time information and plots average active and sleep power for that time interval. Inputs allow users to change the temperature in the power model, which leads to changes in average power consumption.

3.2.1 Interaction with Emulated Software

Emulated software interacts with VarEMU through memory mapped registers. A virtual hardware device maps I/O operations in specific memory regions to VarEMU functions. A command register provides three operations: read, enable

faults, and kill. The *read* operation creates a checkpoint for all VarEMU registers (Figure 3.4). Subsequent reads to register memory locations will return values from the last checkpoint. This allows users to read values that are temporally consistent across multiple registers.

A write to the *enable faults* command register propagates its input value to a variable shared with the VarEMU fault module. A value of 0 means that faults are completely disabled. The implications of a write to the fault register with a value greater than zero depend on the specific implementation of the fault model, but in general such a write means that faults are allowed to happen from this point on.

Finally, a write to the *kill* command register kills the VM and stops emulation. This allows users to systematically finish an emulation session in machines that do not provide the equivalent of a *shutdown* command.

In machines without an operating system (or memory protection), applications may directly interact with the VarEMU memory region. We provide a small library of high level functions that issues the adequate sequence of write/read operations in order to interact with VarEMU. For machines that use the Linux operating system, these operations are embedded into a driver, which also performs per-process time and energy accounting and handles fault status.

3.2.2 Software Interface for Linux

In a multi-process system, it is difficult to attribute energy expenditure to different processes from a global energy meter without system support. Furthermore, it would be very difficult to conduct experiments and evaluate the impact of faults to individual applications in a multi-process system if fault states were allowed to cross process boundaries. For example, if enabling faults in an application led to faults being enabled in kernel code, or in the shell process, the system would

most likely become unstable and/or crash. Nevertheless, a multi-process system typically provides several software conveniences that may not be available in a simpler, OS-less system (e.g. I/O shell, networking stack, remote file copy).

We implemented a series of small extensions to the Linux kernel that allows applications to benefit from its software stack while avoiding the issues described above. First, we extended the process data structure with a new data structure containing VarEMU registers. This field holds fault status and time and energy counters for each process.

When a process is scheduled in, we create a checkpoint by reading all VarEMU registers from hardware. When the process is scheduled out, we create a second checkpoint. Energy and cycles between the schedule in and out events are attributed to the process. Energy and cycles between the out event for the previous process and the in event for the next process are attributed to the operating system. Fault status is part of process context, and hence saved/restored in scheduling events. Thus, enabling faults in one process does not enable faults in other processes or the OS.

Applications interact with the VarEMU driver through a system call interface. A write system call takes two parameters: command and value. Two commands — which map to the corresponding operations in the virtual hardware device — are available: fault and kill. Value is ignored for the kill command. A read system takes two parameters: an integer type and a pointer to a VarEMU data structure (which mirrors the register layout in Figure 3.4). Type can be system, process, or hardware. Read system calls issue the read command to the hardware hardware device, read VarEMU registers, and copy values into the VarEMU data structure provided by the user, according to the type variable. Type can be system (reads counters for the OS), process (reads counters for the current process), or hardware (reads raw hardware counters). A small library of functions aids users in the interaction with the VarEMU driver.

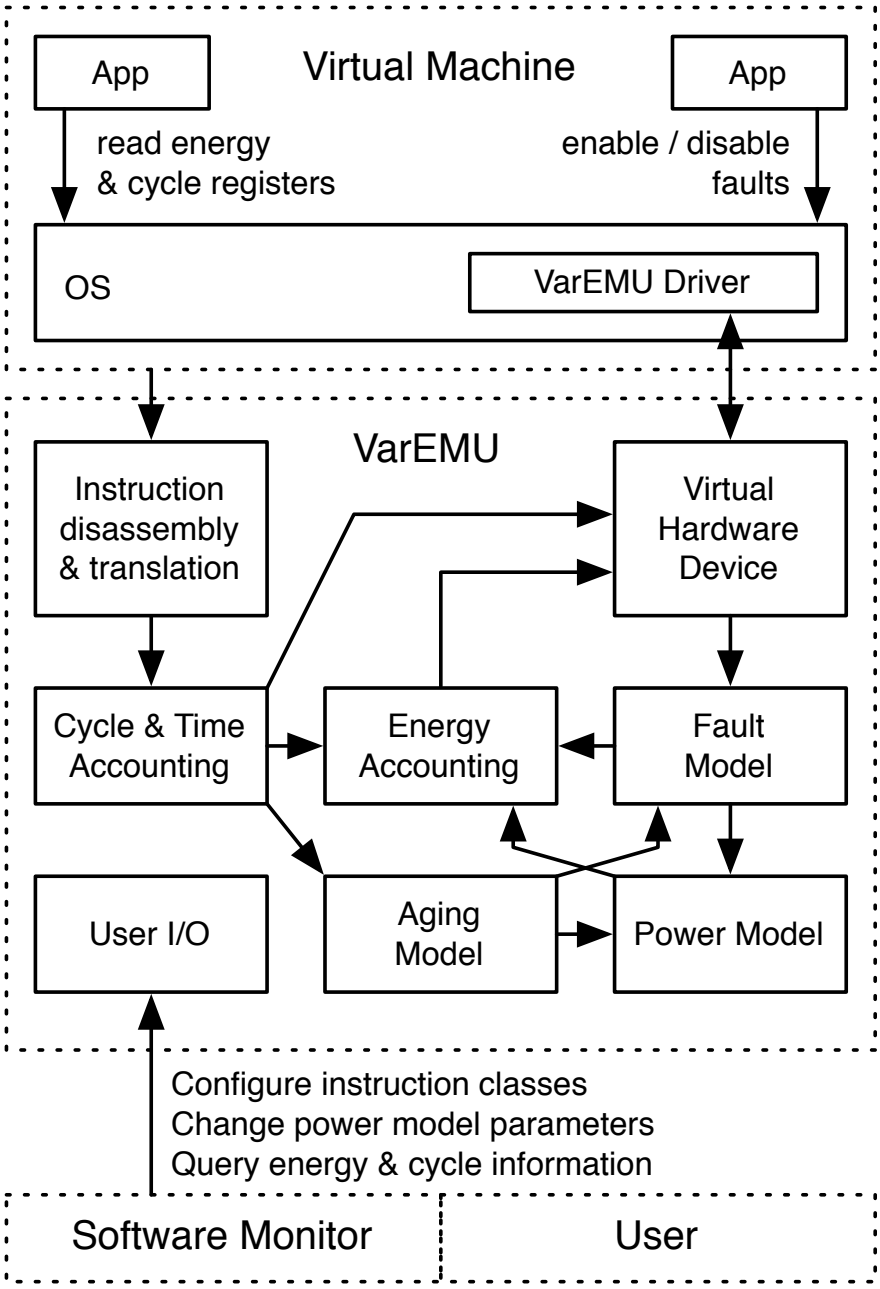
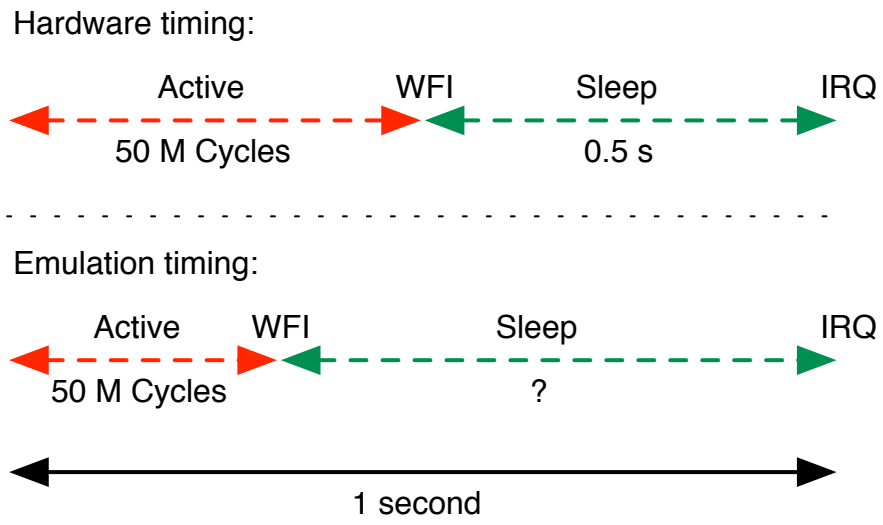
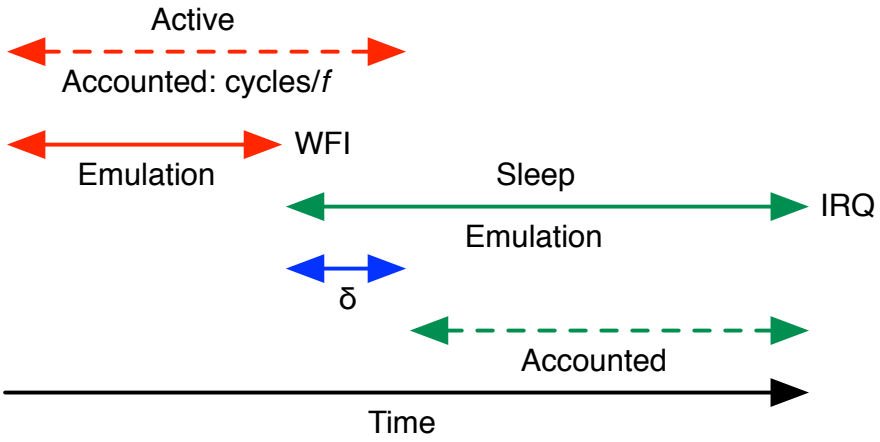


Figure 3.1: VarEMU Architecture



(a) Problem: emulation runs as best-effort, so execution and sleep times do not match hardware.



(b) Solution: keep track of accounted and actual execution times, adjust sleep time interval accordingly.

Figure 3.2: Sleep Time Accounting

```
uint32_t vemu_fault_replace(CPUArchState * env, TranslationBlock* tb)
{
  if (privmode|(vemu_faults_enabled == 0))
    return 0;
  switch(instr_info->opcode) {
    case OPCODE_MUL: {
      int rd = (instr_word >> 16) & 0xf;
      int rs = (instr_word >> 8) & 0xf;
      int rm = (instr_word) & 0xf;
      env->regs[rd] = (env->regs[rm] * env->regs[rs]) | 0x01;
    }; break;
    ...
    default: break;
  }
  return 1;
}
```

Figure 3.3: Stuck-at fault in the multiply instruction

```
typedef struct {
  uint64_t act_time[MAX_INSTR_CLASSES];
  uint64_t act_energy[MAX_INSTR_CLASSES];
  uint64_t cycles[MAX_INSTR_CLASSES];
  uint64_t total_act_time;
  uint64_t total_act_energy;
  uint64_t total_cycles;
  uint64_t slp_time;
  uint64_t slp_energy;
  uint64_t fault_status;
} vemu_regs;
```

Figure 3.4: VarEMU register layout

```
#include <stdio.h>
#include "vemu.h"

int main() {
  vemu_regs hw, sys, proc;
  do {
    usleep(1000000);
    vemu_read(READ_HW, &hw);
    vemu_read(READ_SYS, &sys);
    vemu_read(READ_PROC, &proc);
    printf("Energy: \n");
    printf("hw: %d sys: %u proc: %u sleep: %u\n",
        hw.total_act_energy,
        sys.total_act_energy,
        proc.total_act_energy,
        hw.slp_energy);
    int i, x, y, z, sum;
    vemu_enable_faults(1);
    for (i = 0; i < 100; i++) {
      z = x * y;
      sum = sum + z;
    }
    vemu_disable_faults();
    printf("sum: %d", sum);
  } while (1);
}
```

Figure 3.5: Linux application using VarEMU

Figure 3.5 shows how a Linux application may interact with VarEMU. The `vemu_regs` data structure holds fields for all time, energy, cycle, and fault registers. The main function goes through an infinite loop where it reads and prints out energy values for process, system, and hardware. It then enables faults and goes through a for loop with multiplication and additions. Until faults are disabled again towards the end of the main loop, faults are allowed for this process. This means that, for every instruction configured as susceptible to faults by the user, a call will be issued to the VarEMU fault model. The exact nature of the faults will depend on the fault model implementation and may lead to application crashes (e.g. due to invalid pointers being computed as a result of a faulty add instruction). A fault or crash in this application will not lead to faults in the kernel or in other processes.

While our example application only reads and prints out VarEMU register values, variability-aware applications could use this information to adapt its quality of service based on energy constraints. Likewise, extensions to the OS kernel could use this information to inform scheduling decisions.

3.3 Experiments and Results

This section presents verification and performance results for VarEMU.

3.3.1 Time Accounting Accuracy

VarEMU accounts time on the basis of number of instructions executed, clock frequency, and number of cycles taken by each instruction. In hardware implementations, the number of cycles taken by some instructions may be variable. Because VarEMU relies on an underlying platform of *functional* (not cycle accurate) emulation, this variable timing information is not available to our time accounting module, and instructions are assumed to take a fixed number of cycles

based on their operation code. While this number of cycles may be calibrated to reflect specific platforms and workloads, it is inherently subject to inaccuracy.

To quantify the accuracy of time accounting in VarEMU, we compare execution times in hardware with execution times reported by VarEMU for different applications. For each application tested, we follow the same sequence of events: 1) a GPIO pin is raised, 2) a VarEMU *read* command is issued 3) the main body of the application is executed, 4) the GPIO pin from step 1 is lowered, and 5) a new *read* command is issued. Because both the GPIO write and the VarEMU read command can be implemented with a single "write" instruction (in systems without an OS), there is only one instruction difference between the two. By connecting the GPIO pin to an oscilloscope and measuring is logical high period, we can quantify execution time in hardware.

For this evaluation, we used the LM36965 model Cortex-M3 processor by Texas Instruments. When running in hardware, interaction with VarEMU is replaced with equivalent read/write operations in a reserved area in memory. GPIO operations have no effect in QEMU, but are still accounted for (i.e. a read or write instruction is executed). To check against cumulative errors, we ran a varying number of iterations for each application.

Figure 3.6 shows VarEMU time accounting accuracy for different applications. Accuracy is defined as the ratio between actual execution time in hardware and execution time reported by VarEMU. We calibrated the number of cycles per instruction using the "empty loop" application, and hence that application has the highest accuracy. For all other applications, accuracy is better than 96%, and does not increase with longer execution runs. In future work, we intend to increase this accuracy by performing deeper inspection of instruction words (e.g., in Cortex-M3 cores, some instructions take more or less cycles depending on which registers are used), and by performing basic bookkeeping on branches and load/store instructions to estimate pipeline bubbles.

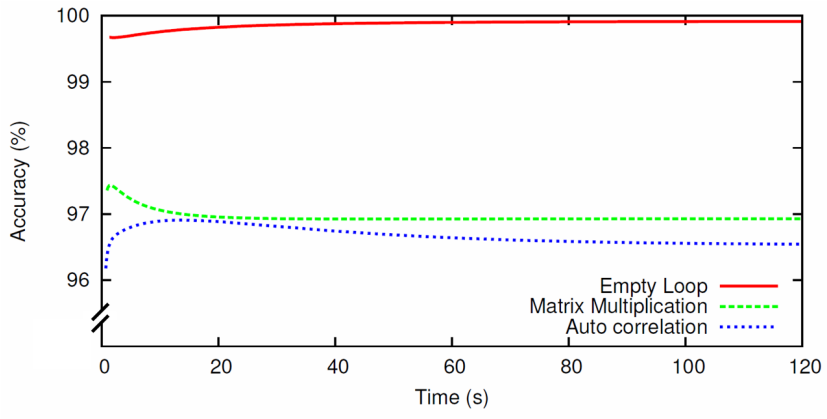


Figure 3.6: Time Accounting Accuracy

3.3.2 Runtime Overheads

Every time an emulated instruction is executed a call is made to the VarEMU module that performs cycles and time accounting. Periodically, the cycle counting module makes calls to the aging module. If an instruction is susceptible to errors, its translated code is augmented with calls to the error module. Finally, every time a query is issued for the energy counters, or whenever a variability model parameter (e.g. temperature) changes, the power model is called.

On the emulated software system, the Linux module for VarEMU performs per-process energy and time accounting. Every time a process is switched in or out, a read command is issued to the VarEMU virtual hardware module, and all VarEMU registers are copied. When an OS is not available, the standalone VarEMU library performs the same function.

To quantify the various runtime overheads of VarEMU, we compare runtime performance of software under VarEMU with its equivalent performance under the vanilla version of QEMU. We measured the relevant performance metrics (e.g. time-to-completion, throughput) of various software applications. Table 3.1 presents the resulting average of each application's metric over 10 runs.

The overhead of VarEMU over the vanilla version of QEMU is dependent on

App	Unit	Vanilla QEMU	VarEMU	Overhead	Kernel	Overhead	Total Overhead
Dhrystone	p/sec.	259304	102536	150%	98352	4 %	164 %
Whetstone	MIPS	14.2	5	180%	4.8	4 %	196 %
null syscall	μs	12.4	13.5	9%	13.5	0 %	9 %
context switch	μs	61	75.6	24%	88.3	17 %	45 %
dd /dev/zero	s	0.98	1.43	46 %	1.49	2 %	49 %
JPEG	s	0.9	1.3	45 %	1.31	1 %	46 %
MP3 (lame)	s	19.1	57.3	200 %	57.4	0 %	200 %

Table 3.1: Runtime overheads for VarEMU and the VarEMU kernel extensions

workload. This is due to the fact that some emulated instructions (e.g. integer arithmetic) translate very efficiently into native instructions, while others (e.g., load/stores, branches) have higher emulation overhead. Because VarEMU adds a function call with constant execution time to each instruction, for very efficient instructions the VarEMU extensions become a significant part of total execution time. For less efficient instructions, VarEMU overhead is relatively smaller. For our test applications, best-case overhead was 9%, and worst-case 200%.

The overhead of the Linux kernel extensions for VarEMU also depends on workload. Bookkeeping is performed for every process switch, and therefore the context switch operation has the highest overhead, at 17%. For the other applications in our test set, the overhead is at most 4%. Total combined overhead for VarEMU, including emulation and kernel overheads, ranged from 9% to 200% for our test applications. Since QEMU (in combination with a fast host system) provides faster than real-time emulation for many of its target platforms, this overhead is manageable, and much smaller than that of other simulation alternatives such as cycle-accurate simulators.

CHAPTER 4

Variability-Aware Duty Cycling

Wireless embedded sensing systems employ a variety of power management techniques to achieve system lifetime objectives [RGS06]. A particularly common technique is duty cycling [DGA05], where the system is by default in a sleep state and is woken up periodically to attend to pending tasks and events. A higher duty cycle rate typically translates into higher quality of service [ZSB10]. A system with higher duty cycle may, for example, sample sensors for longer intervals or at higher rates, increasing data quality. A typical application-level goal is to maximize quality of data through higher duty cycles, while meeting a lifetime goal. While duty cycles in embedded sensing applications range from below 1% in Car-Park management [BOO06] and CargoNet [MMF07], to greater than 50% in VigilNet [HVY06], often the duty cycle ratio is extremely small ($<< 1\%$), and the energy consumed by the platform during the sleep state accounts for almost all ($> 99\%$) of the energy consumption.

Significant variability in power across nominally identical instances and across temperature is already present in contemporary embedded processors, and is expected to increase in subsequent semiconductor manufacturing processes. Duty cycling is particularly sensitive to variations in sleep power at low duty cycling ratios. Variability implies that any fixed, network-wide choice of duty cycle ratio that is selected to ensure desired lifetime needs to be overly conservative and result in lower quality of sensing or lifetime.

In systems where computation is constrained by limited energy reserves and

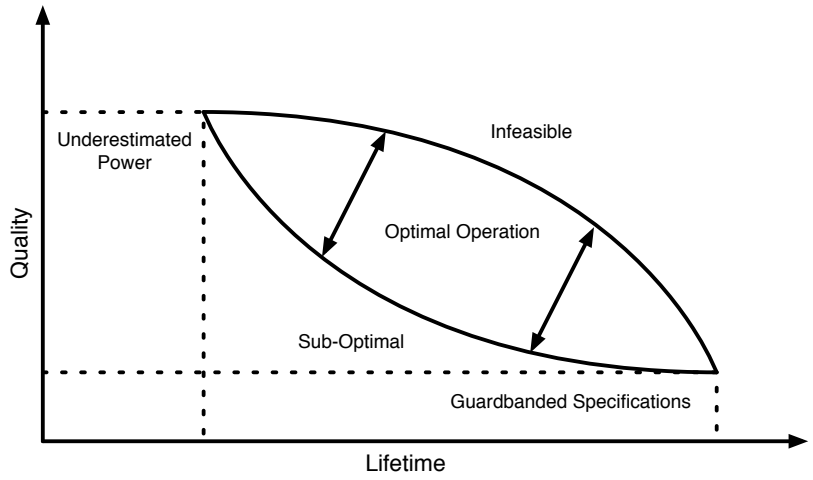


Figure 4.1: Potential results of variability in terms of system quality and lifetime

where a long overall system lifetime is desired, maximizing duty cycle (and hence quality) of a given application subject to these constraints is both challenging and an important step towards achieving high quality deployments. Underestimation of system power consumption can lead to a reduction in lifetime which will eventually impact quality of service, while guardbanding against worst-case power consumption by using overly conservative estimates can reduce application quality for the entirety of the lifetime. The potential solution space is shown in Figure 4.1, where the optimal solution is one that maximizes quality without decreasing lifetime.

Consider for example an application running on the Atmel SAM3U that periodically wakes up, samples a sensor, sends the result to a forwarding module (e.g. for storing, sending to the network), and goes into sleep mode. We assume that the sampling task will complete within 10 ms with a 4 MHz clock in active mode. For the sake of clarity, we also assume that the system and the forwarding module take negligible time and power to complete their operations. The performance requirement is a desired lifetime of one year using one AAA battery (850 mA-h) operating at 1.8V. As shown in Figure 4.2, across ten hardware samples and over a temperature range of $0 - 50°C$, the worst case duty-cycle to achieve the de-

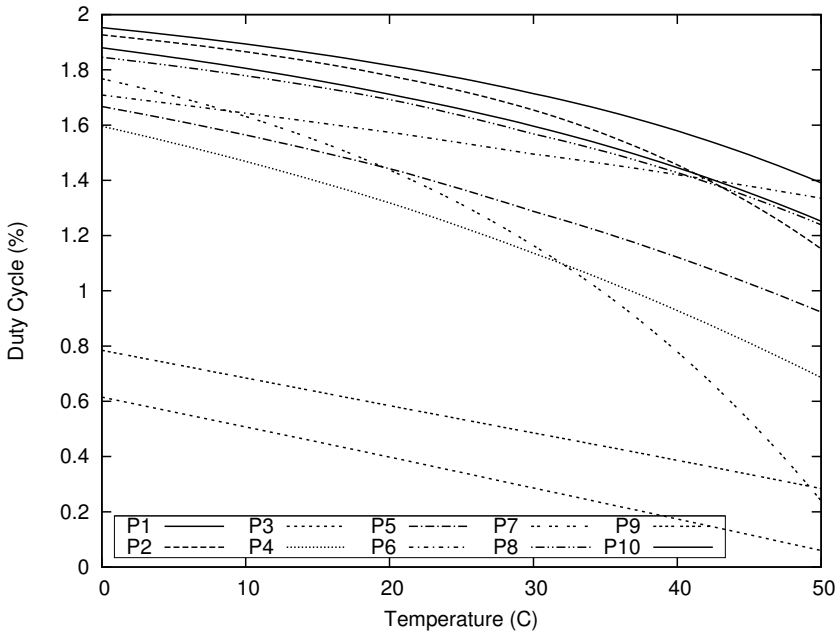


Figure 4.2: Allowable duty cycle across SAM3U instances and temperature

sired lifetime is approximately 0.1%, which results in a sleep duration of 9990 ms for every active period of 10 ms. The best case duty-cycle is approximately 2%, which results in a sleep duration of 490 ms and approximately $20x$ more sensor data being collected.

4.1 Duty Cycle Scheduling

A duty cycle schedule indicates the activity rate of a system at any point in its lifetime. An optimal duty cycle schedule maximizes the active time of the system across its desired lifetime, given an energy constraint. If there is no variability in power consumption, the optimal duty cycle schedule can be uniform across the lifetime of the system.

Given an energy budget of E Joules, a lifetime of L seconds, and invariable constants for active and sleep power consumption P_A and P_S Watts, the maximum allowable allowed duty cycle DC is given in (4.1). The values of P_A and P_S can

be typically obtained from the processor datasheet. We henceforth refer to this as ***datasheet-based duty cycle***.

$$P_A \cdot DC + P_S \cdot (1 - DC) = \frac{E}{L}$$
$$DC = \frac{E - L \cdot P_S}{L \cdot P_A - L \cdot P_S} \tag{4.1}$$

4.1.1 Variable Power Consumption

When instance and temperature-dependent variation is taken into consideration, the ***worst-case uniform duty cycle*** can be found by applying the worst-case active and sleep power consumption across all instances and operating temperature range as constants P_A and P_S in Equation (4.1). We henceforth refer to this as ***worst-case duty cycle***.

With prior characterization, active and sleep power can be expressed as functions of temperature $P_A(T)$ and $P_S(T)$. Additional peripheral components used while in active mode, such as radios or sensors, can be added to total active mode power as constants or functions, depending on whether power variation is present or not. If the temperature profile is known (or can be learned) for the lifetime of the system, temperature can be expressed as a frequency distribution. For a known operating temperature profile and a given processor instance, the problem of finding an ***optimum duty cycle*** can be formulated as a linear program. Given the expected frequency distribution of (discretized) temperatures across the lifetime of the application, the optimum duty cycle at each temperature T, DC_T is given by Equation (4.2):

$$\arg\max_{DC_T} \sum_{T=T_{min}}^{T_{max}} DC_T f_T \tag{4.2}$$

$$\text{such that} \sum_{T=T_{min}}^{T_{max}} f_T \cdot (P_A(T) \cdot DC_T + P_S(T) \cdot (1 - DC_T)) \leq \frac{E}{L}$$

$$DC_{min} \leq DC_T \leq DC_{max}$$

$$T_{min} \leq T \leq T_{max}$$

where f_T is the relative frequency of temperature T across the lifetime L, assuming discretized temperature bins. DC_{min} and DC_{max} are the minimum and maximum duty cycles allowed for the application. The maximum duty cycle constraint can be used to limit duty cycles when increasing duty cycle beyond a given rate would bring no further increase to quality of service.

4.1.2 Variability-Aware Uniform Duty Cycle

Assuming a uniform duty cycle $DC_T = DC^*$ independent of temperature, we can determine DC^* that satisfies the constraints given in Equation (4.2).

$$DC^* = \min\left[\gamma, DC_{max}\right] \tag{4.3}$$

$$\text{where } \gamma = \frac{E - L \cdot \sum_{T=T_{min}}^{T_{max}} P_S(T) \cdot f_T}{L \cdot \sum_{T=T_{min}}^{T_{max}} (P_A(T) - P_S(T)) \cdot f_T}$$

Moreover, it can be shown that when $P_A(T) - P_S(T)$ is constant across all T, DC^* is the ***uniform duty cycle*** that optimizes the linear program in (4.2). We observed this to be *practically* true under nominal operating temperatures for the current generation microprocessors, like the Atmel SAM3U, because (i) their sleep power consumption $P_S(T)$ is much less than active power consumption $P_A(T)$, and (ii) the $P_A(T)$ is effectively constant as active mode leakage power is insignificant for their fabrication technology, and switching power variation across

normal temperatures is small. Henceforth, whenever we refer to ***variability-aware duty cycle***, we are referring to (4.3).

4.1.3 Reactive Duty Cycle

Allowable duty cycle rates can also be found dynamically through measurements or estimations of past power consumption, given total energy capacity at the start of lifetime. Energy consumption can be directly measured with dedicated monitors [MHY06], inferred from remaining battery capacity [LMM07], or through variability-aware models that estimate energy expenditure by measuring conditions that affect power consumption, e.g. temperature and activity rates.

In a reactive model, duty cycle can be dynamically determined at time t as a ratio of duty cycle at time $t-1$, according to energy spent from time $t-1$ to time t, and remaining energy in the system. Remaining energy at time t is given by $E_t = E - \sum_{i=0}^{t-1} P_i$, where E is the total energy capacity and P_i is power estimated or measured at time i. An example of a ***reactive duty cycle*** adaptation model is given in (4.4).

$$DC_t = \frac{E_t \cdot DC_{t-1}}{(E_t - E_{t-1}) \cdot (L - t)} \tag{4.4}$$

The reactive model in (4.4) assumes that the power consumption rate for the previous time period is indicative of the power consumption for the remainder of lifetime of the system. While more complex models could incorporate longer histories, any reactive model will depend on accurate measurement of past energy consumption or estimation of remaining battery energy. While systems with dedicated power monitors have been explored in the literature [MHY06], it is acknowledged that their integration in low power sensing platforms would likely result in prohibitive cost overhead. Hence, most systems rely on estimations of remaining battery capacity to infer energy consumption [LMM07]. Battery ca-

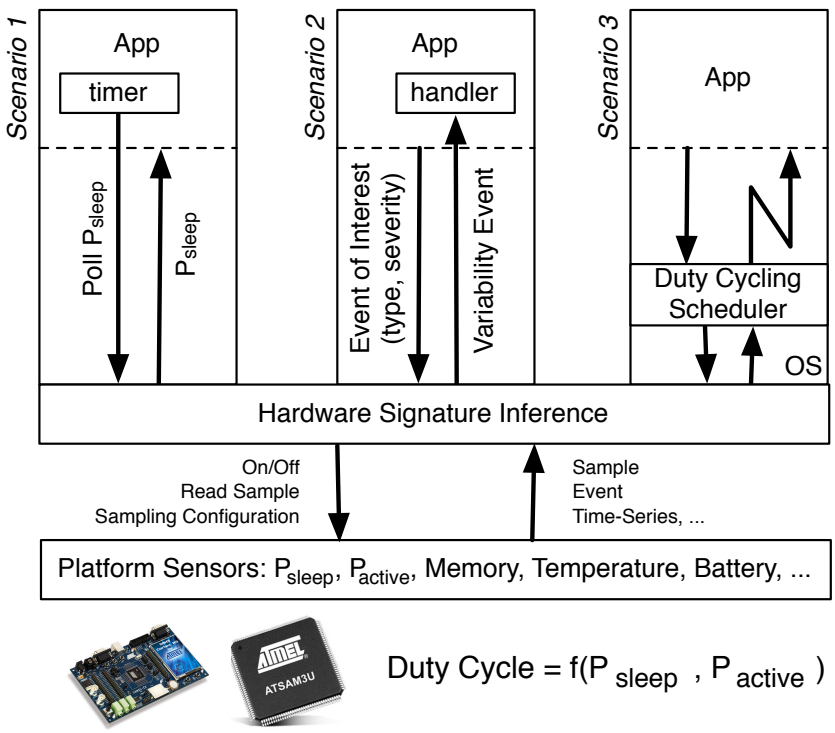


Figure 4.3: Designing a software stack for variability-aware duty cycling

pacity estimation is a research issue in itself and subject to inaccuracies. In target systems with long lifetimes (e.g. greater than a year), in which the energy consumed in one hour might be less than 0.01% of total battery capacity, this makes short-term adaptation problematic. We therefore do not use this method.

4.2 Variability-Aware Duty Cycle Software Adaptation with TinyOS

In order to maximize the sensing quality in the presence of power variation, an opportunistic sensing software stack can help discover and adapt the application duty cycle ratio to the sleep mode power variations across parts and over time. The run-time system for the opportunistic stack will have to keep track of changes in hardware characteristics and provide this information through interfaces accessible to either the system or the applications. Figure 4.3 shows several different ways

such an opportunistic stack may be organized; the scenarios shown differ in how the sense-and-adapt functionality is split between applications and the operating system. Scenario 1 relies on the application polling the hardware for its current "signature". In the second scenario, the application handles variability events generated by the operating system. In the last scenario, handling of variability is largely offloaded to the operating system.

Using architecture similar to that of scenario 3 in Figure 4.3, we implemented a prototype variability-aware duty cycling framework in TinyOS. Application modules specify to the scheduler a range of acceptable duty cycling ratios, and the scheduler selects the actual duty cycle based on run-time monitoring of operational parameters, and a power-temperature model that is learned off-line for the specific processor instance. While this approach is potentially less flexible than the ones presented by scenarios 1 and 2, it simplifies application development by abstracting the underlying complexities of the variability signature model.

TinyOS [LMP05] differs from traditional operating systems in that it is *event-based*. Applications respond to events (e.g. interrupts from hardware, incoming radio messages) with event handlers. These handlers should typically complete within a few hundred processor cycles. To execute long running computations, applications post tasks, which work as deferred function calls. Each task runs to completion on a scheduler loop. Whenever the system has no tasks to schedule, it puts the processor in sleep mode, waiting for the next interrupt, which will trigger new event handlers and potentially new tasks. The event handler / background tasks model of TinyOS naturally lends itself to duty cycled systems: event handlers and tasks represent active periods, empty scheduler queues lead to inactive periods. Nevertheless, there's no explicit support for discovering and adapting duty cycle in TinyOS.

We introduce a new ***Duty Cycle Scheduler*** to TinyOS. Figure 4.4 shows our system architecture. A hardware signature inference module provides power

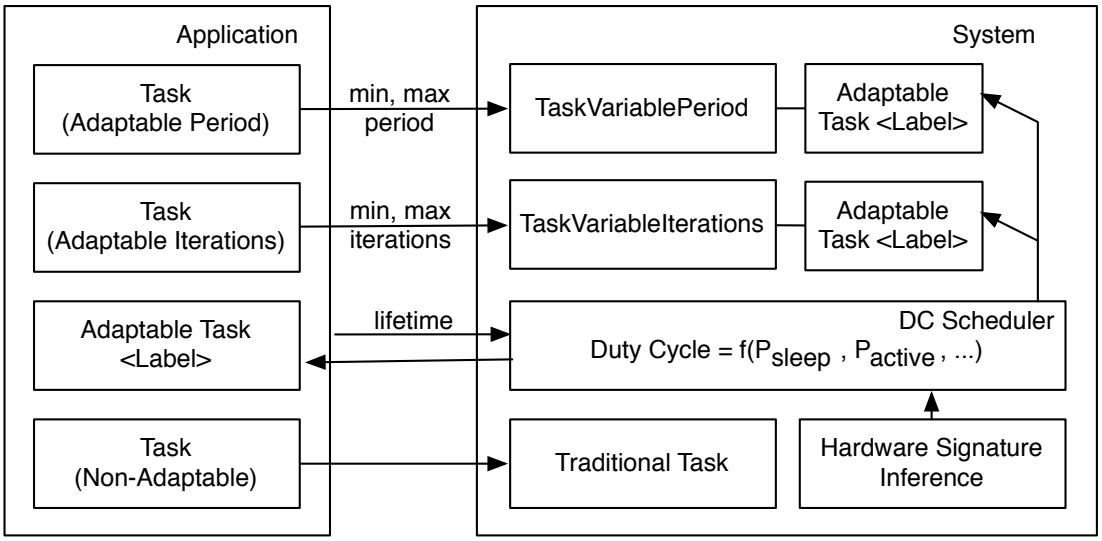


Figure 4.4: System architecture for variability-aware duty cycle scheduling in TinyOS

vs. temperature curves for each processor instance. While in our TinyOS implementation we assume that these curves are pre-characterized, extensions to this module could feature online learning through dedicated power meters and take other variability vectors such as aging into account.

The scheduler determines an allowable duty cycle based on: (i) sleep and active power vs. temperature curves provided by the hardware signature inference module, (ii) temperature profile for the application, which can be pre-characterized or learned dynamically, (iii) lifetime requirement, (iv) battery capacity, and (v) the variability-aware duty cycle formulation presented in Equation (4.3) in Section 4.1.2.

In order to maintain compatibility with existing system code that makes liberal use of standard tasks, we introduce a new class of ***Adaptable Tasks***. These tasks respond to events from the Duty Cycle scheduler that informs them of their allowable duty cycle compared to current duty cycle. However, the system does not *enforce* adaptation of these tasks. These are assumed to adapt according to the duty cycle change event from the scheduler.

For standard applications, we provide two additional classes of tasks, which implement two common adaptation scenarios: tasks with variable iterations and tasks with variable period. For the first class, the programmer provides a function that can be invoked repeatedly a bounded number of times within each fixed period. For the second class of tasks, the application programmer provides a function representing task functionality that is invoked once within each variable but bounded period of time. Internally, each of these tasks uses an adaptable task and unique identifier. The system adjusts the number of iterations or period of the task based on the allowable duty cycle informed to its underlying adaptable task.

To adapt duty cycle, the system first needs to account active and sleep time for the entire application. Active time is divided into fixed, non-adaptable computation time C_f (for traditional tasks and interrupt handlers) and adaptable computation time $C_a(i)$ (for each adaptable task i). For each task activation, the system registers timestamps and accumulates computation time in the appropriate counters. This incurs in a small overhead to task activations quantified in section 4.2.1.5.

For every accounting period τ, the system compares total computation time $C_\tau = C_f + \sum_1^N C_a(i)$, where N is the number of adaptable tasks in the system, and allowable active time $C_{DC} = \tau \times DC$, where DC is the allowable duty cycle for the node. To allow adequate timing estimations with low duty cycles, the accounting period τ is large enough to encompass several active/sleep cycles for each task (τ = 10 minutes in our implementation). If $|C_\tau - C_{DC}| <= \delta$, where δ is an arbitrary tolerance, the system has converged to the allowable duty cycle. Otherwise, each adaptable task is assigned a new allowable computation time $C_{DC}(i)$ obtained by dividing available active time equally between all adaptable tasks. The new computation time is informed to the tasks in the form of a ratio to the previous time $C_a(i)$ through an event. The ratio is used to adjust the number

of iterations or period duration of tasks.

4.2.1 Evaluation

To evaluate the duty cycle scheduling methods, we make use of a common scenario in embedded sensing: a long running duty cycled application with a limited energy source (battery), which periodically becomes active to perform sensing/processing tasks, and subsequently returns to a low-power sleep mode until the next period.

We assume an application which, when active, uses only the main processor running at 4MHz, and when in sleep mode, disables all peripherals except for a low-power wake-up timer. Table 4.1 summarizes the results we present in this section. Active and sleep power are obtained from the characterization model and measurements presented in Chapter 2. Figure 4.12 is based on technology projections. All other results are based on the models fitted to SAM3U measurements. The temperature profile is based on hourly temperature data from the National Climactic Data Center [US14], and specified for each result.

Fig.	Type of Result	Variable
4.6	Duty Cycle Schedules	Time
4.9	Improvement with Var-Aware DC	Temperature profile
4.10	Lifetime with Datasheet-Based DC	Temperature profile
4.11	Improvement with Variability-Aware DC	Battery Capacity
4.12	Improvement with Variability-Aware DC	Technology

Table 4.1: Summary of Results for Variability-Aware Duty Cycling

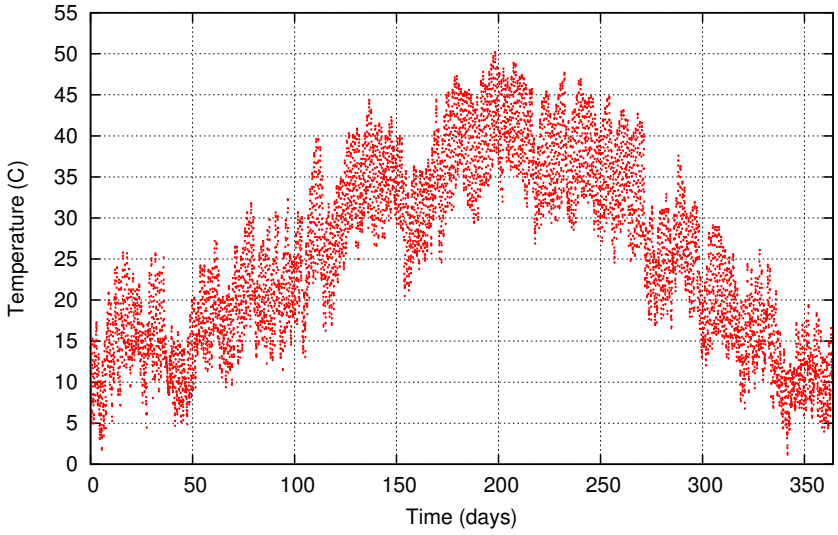


Figure 4.5: Weather profile for Death Valley, CA, 2010.

4.2.1.1 Comparison of Duty Cycle Scheduling Methods

We first compare duty cycle schedules resulting from the worst-case, datasheet-based, and variability-aware methods discussed in Section 4.1. We assume an energy supply of 5400 mA-h from two AA batteries, a lifetime of one year, and temperature profile based on the location of Stovepipe Wells, CA (Death Valley National Park), which has extreme seasonal and daily temperature variations and hence clearly illustrates the differences between the duty-cycling regimes. Figure 4.5 shows the temperature trace used for this experiment.

Figure 4.6 shows the DC schedules across the lifetime of an application for all processor instances. It shows that the duty cycle resulting from datasheet power specifications usually does not meet the required lifetime, as the specification is not guardbanded enough. Determining a completely pessimistic sleep power specification is very difficult since leakage distribution has a long tail. The worst-case duty cycle is exceedingly low, as it assumes the worst-case and power across all instances and the entire temperature range to which the application experiences, and hence leaves out untapped energy resources. The variability-aware duty cy-

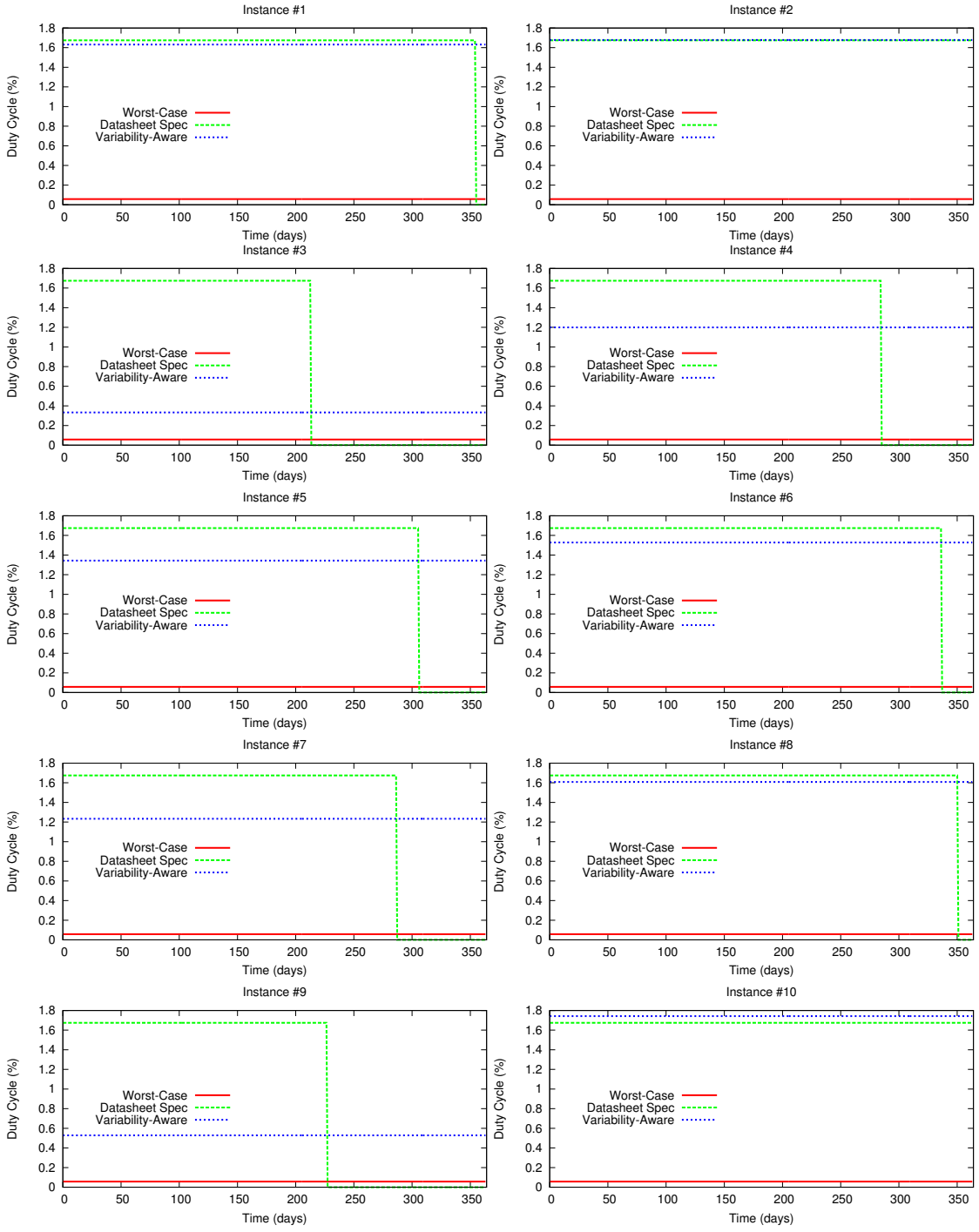
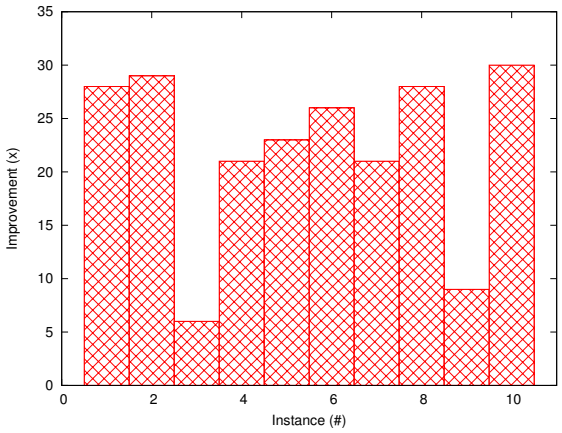
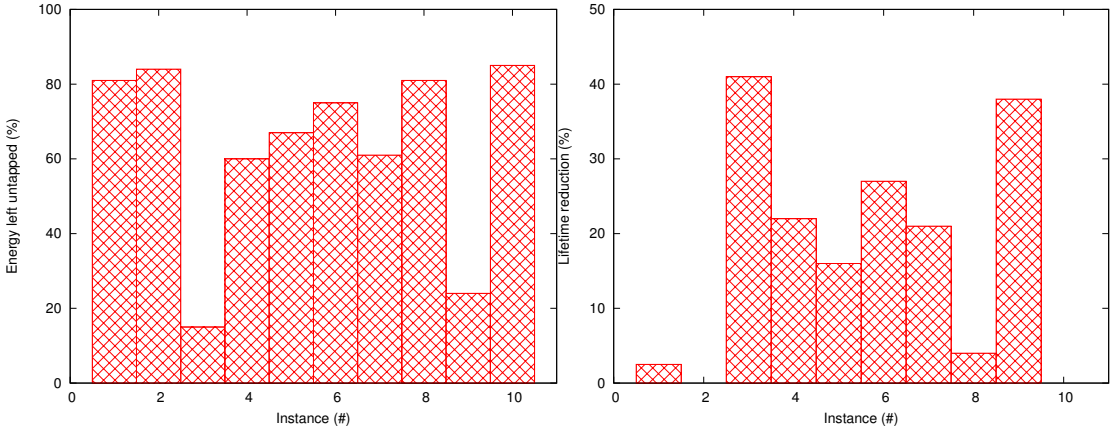


Figure 4.6: Duty cycle schedule for all instances.



(a) Improvement of variability-aware duty cycle over worst-case duty cycle



(b) Energy left untapped with worst-case duty cycle

(c) Lifetime reduction with datasheet-based duty cycle

Figure 4.7: Results from duty cycle regimes across all instances

cle maximizes active time, constrained by lifetime requirements and application temperature profile. Figure 4.7 shows the per-instance improvement of variability-aware duty cycle over worst-case duty cycle, energy left untapped with worst-case duty cycle, and lifetime reduction with datasheet-based duty cycle. On average, we found a 22x improvement in active time with the variability-aware DC over the worst-case DC, 63% of energy potential left untapped by the worst-case duty cycle, and 15% reduction in lifetime with duty cycle based on datasheet specifications.

4.2.1.2 Temperature Profile

Next, we use the same energy and lifetime scenario described above for 140 locations with different temperature profiles. Figure 4.8 shows the average, minimum, and maximum temperature for all the test locations.

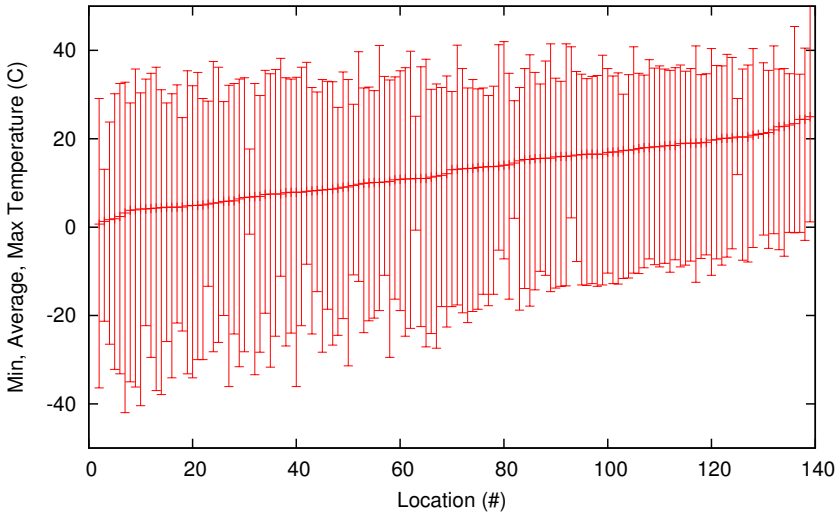


Figure 4.8: Temperature profile for test locations.

Figure 4.9 shows the average improvement of variability-aware duty-cycle compared to worst-case duty cycle. There is an exponential relation of improvement with temperature. This results from the exponential nature of leakage power with temperature: as maximum temperature increases, leakage power increases exponentially, and hence the worst-case duty cycle is exponentially worse than the variability-aware duty cycle. For any given maximum temperature, improvement also depends on temperature distribution: temperature profiles with higher maximum temperatures benefit more from the variability-aware scheme, as the worst-case power becomes progressively worse with higher temperatures. The average improvement across all temperature profiles is 6.4x.

Figure 4.10 shows lifetime reduction in days when duty cycle is determined based on the datasheet specification. There is a linear dependence between lifetime and average temperature. As average temperature increases, the difference

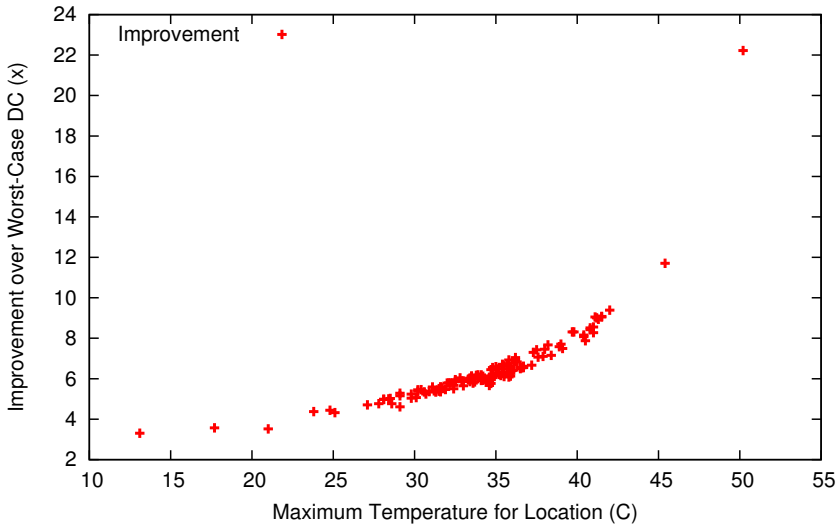


Figure 4.9: Improvement over worst-case duty cycle for test locations.

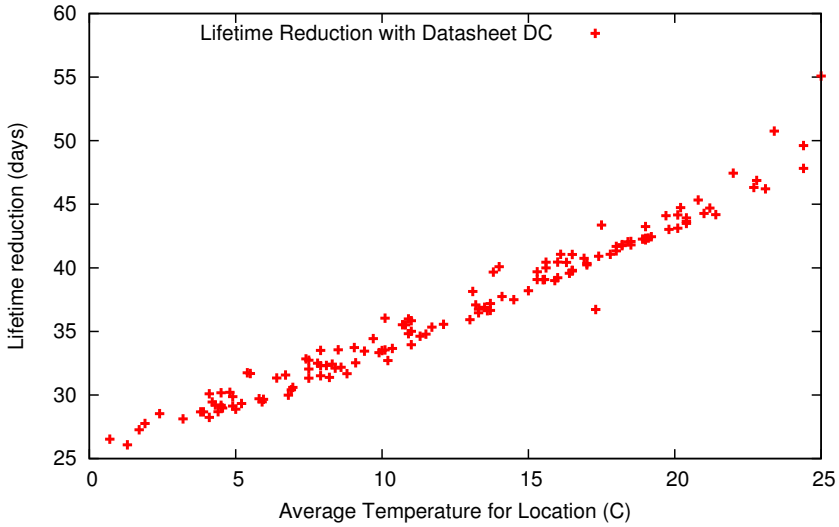


Figure 4.10: Lifetime reduction with DC based on datasheet.

between actual and spec power increases, and hence lifetime decreases. Across all temperature profiles there is an average lifetime reduction of 37 days for a lifetime of one year.

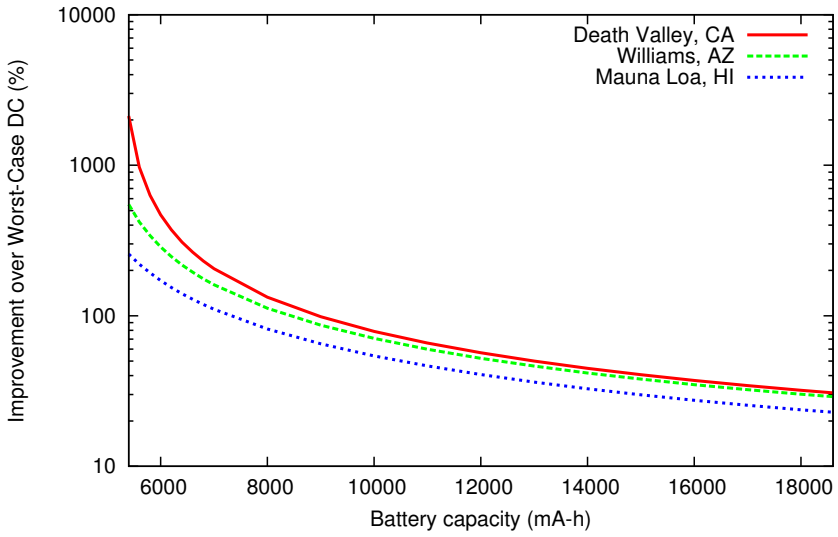


Figure 4.11: Improvement over worst-case DC across battery capacities.

4.2.1.3 Battery Capacity

Figure 4.11 shows the average percentile improvement of a variability-aware duty cycle schedule (4.3) over the worst-case duty cycle for different battery capacities. Each curve shows improvement with a different temperature profile. Death Valley, CA has the most extreme temperatures, and hence greater improvement. Mauna Loa, HI has low average temperatures and little temperature variation, and hence benefits the least from the variability-aware scheme. Williams, AZ represents the average case.

This plot shows that the variability-aware duty cycling regime is more advantageous for applications with smaller duty cycles. For "small" batteries (5400 mA-h, or 2 AA batteries), improvement is more than 100% for all temperature profiles. The improvement for very large batteries (20 A-h), improvement is between 20% and 30%, depending on temperature profile. The worst-case duty cycle with a 20 A-h battery is more than 5% for all temperature profiles. This suggests that, for current embedded fabrication technologies, this scheme is beneficial for small (less than 5%) duty cycles.

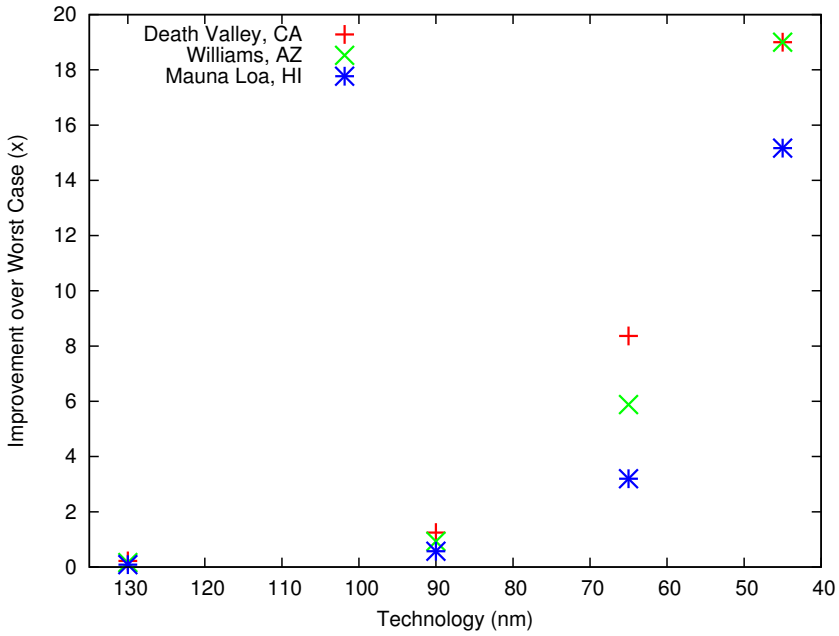


Figure 4.12: Projection of improvement with scaling of technology.

4.2.1.4 Technology Projections

Figure 4.12 shows the improvement of the variability-aware duty cycle over worst-case duty cycle with technology scaling as per the model presented in Chapter 2. As with the previous result, we show three curves with different temperature profiles. For each point in the curve, we simulate a battery capacity large enough to support a worst-case duty cycle of 5%. As noted in the previous results, with a worst-case duty cycle of 5%, there are only marginal benefits of a variability-aware duty cycle schedule for current technology characteristics (130nm). As technology progresses and the ratio between active and sleep power decreases [PSB05], variability-aware duty cycling regime shows considerable benefits even for this relatively high duty cycle. At 45nm, the improvement saturates at 19x for two of the temperature profiles (duty cycle cannot be higher than 100%).

4.2.1.5 Runtime Overheads

We profiled our duty cycle scheduler implementation for the SAM3U processor, running at 4 MHz. Compared to the base TinyOS scheduler, our implementation requires an additional 20 bytes of RAM memory. Each adaptable task instance uses 5 bytes of memory, and the TaskVariablePeriod and TaskVariableIterations abstractions require an additional 4 bytes per task. This overhead is well within the capacity of low-end sensor nodes (typically > 4KB RAM).

Compared to basic TinyOS tasks, each adaptable task activation has an overhead of 10μs. As a point of comparison, the typical startup and conversion times for the ADC in this platform are 30μs and 20μs, respectively. Finally, the uniform variability-aware duty cycle control module, which runs periodically every 10 minutes, completes within $20(N+1)\mu$s, where N is the number of adaptable tasks in the system. This time is required to distribute available active time across all adaptable tasks, and to determine the rate of activity (i.e. period, number of iterations) of each task. When the system becomes stable, i.e., when all the tasks reach their allowable duty-cycle, the uniform variability-aware duty cycle control module completes within $20\mu s$. The runtime overhead of our simple duty cycling abstractions is comparable to related solutions [ZEL02, LMM07].

4.3 Duty Cycle and Utility optimization with VaRTOS

In our TinyOS implementation we use a simple scheduling mechanism, typical of small embedded sensor operating systems. When multiple adaptable tasks are used alongside traditional, non-adaptable tasks, the burden of adaptation to the allowable duty cycle is distributed equally across all adaptable tasks. This hinders the use of this system with multi-task applications where tasks may have variable importance and potentially competing goals. In this section we present a real-time duty cycle scheduler that overcomes this limitation by assigning energy

resources to tasks according to reward and priority functions. This is accomplished by introducing task *knobs*—task-specific expressions of quality and power elasticity – in an architecture we call VaRTOS, the Variability-aware Real-Time Operating System. VaRTOS is implemented as a as a series of extensions to the FreeRTOS real time operating system [Fre13], though its architecture could be applied to other embedded operating systems as well. FreeRTOS provides typical operating system abstractions such as preemptive scheduling of multiple tasks, synchronization primitives, and dynamic memory allocation with low overhead and small memory footprint.

Like our TinyOS stack, VaRTOS is a realization of Scenario 3 in Figure 4.3. Tasks inform the operating system of their duty cycle bounds, and the scheduler adjusts task activations in order to converge to a system-wide duty cycle that meets lifetime constraints. In order to tune the task-specific duty cycle ratio, we introduce the notion of task knobs. In practical terms, a task knob is a variable that determines either (1) the computation time of a task or (2) the frequency with which a task is activated. Knobs can thus be seen as a generalization of the adaptation mechanisms presented in Figure 4.4: variable iterations lead to variable computation time, and variable periods lead to variable activation frequencies. We argue that a large portion of tasks found in embedded applications will fall in one of these two classes, and those that require both frequency and period modulation can often be divided into two legal subtasks coupled with inter-process communications. For example, tasks that fall under class 1 include variable length sensing tasks, tasks that listen for inbound communication, and variable length processing chains. Those that fall under class 2 include variable frequency transmission, variable frequency sensor sampling, time synchronization handshaking, control and actuation events.

Unlike our TinyOS stack, task knobs in VaRTOS are also used to denote task *utility*. We define task knobs as k_i, such that increasing k_i will increase duty

cycle d_i, and consequently utility u_i. Task knobs are created by passing a variable address to the OS, allowing direct manipulation of knob values by an optimization routine. In addition, the developer specifies a minimum and maximum knob value, $k_{i,min}$ and $k_{i,max}$. The value $k_{i,min}$ specifies the minimum value of k_i that yields a nonzero utility. Below this value, a task offers no utility. The value $k_{i,max}$ specifies a value after which increasing k_i further will yield no added utility. As an example, a radio transmission task may be useless if it does not meet a certain latency requirement, but usefulness may plateau at some frequency governed perhaps by the physics and time response of the event being sensed. A priority parameter p_i allows users to arbitrarily scale utility.

Tasks are issued in VaRTOS with a modified version of FreeRTOS's task creation function with additional parameters to account for task knobs and bounds. Figure 4.13 illustrates task implementation and issuing in VaRTOS, where (a) is the task creation API, (b) is a task where the knob changes task computation time, (c) is a task where the knob changes task activation frequency, and (d) is the application's main routine that creates the tasks. The sensor task in (b) acquires a variable number of sensor samples, leading to variable execution time. The radio task in (c) takes constant execution time, but has variable frequency. The knob bounds in (d) determine the minimum and maximum execution time and frequency of the sensor and radio task, respectively.

4.3.1 Mapping task knobs to duty cycle and utility

The function mapping k_i to duty cycle d_i is not known *a priori*, but is assumed linear and learned through regression at runtime. Dividing active time accumulated per task by a fixed supervisory time interval t_{super} yields task-specific duty cycle ratios, d_i. Active time per task $t_{a,i}$ is measured using hardware timer snapshots at the context swap level. The mapping from k_i to d_i for each task is arrived at by systematic perturbation of k_i within the range $[k_{i,min}, k_{i,max}]$. The knob value k_i is

```
xTaskCreate( TaskFunction, "name", StackSize, Priority, &TaskHandle,
   &TaskKnob, k_min, k_max, p_i);
```

(a) Task Creation API

```
static void SensorTask( void *pvParameters ) {
  // ...
  for( ;; ) {
    vTaskDelayUntil( &xLastExecutionTime, SENSOR_DELAY );
    sensor_val_sum = 0.0;
    sensor_num = 0;
    for( i=0; i<sensor_knob; i++ ){
      sensor_val_sum += getSensorValue();
      sensor_num++;
    }
    if( sensor_num > 0 ){
      sensor_val_avg = sensor_val_sum/((float)sensor_num);
      printf("avg: %f \n", sensor_val_avg );
    }
  }
}
```

(b) Task with Variable Execution Time

```
static void RadioTask( void *pvParameters ) {
  // ...
  for( ;; ) {
    radio_delay = 1000*(1000/portTICK_RATE_MS)/radio_knob;
    vTaskDelayUntil( &xLastExecutionTime, radio_delay );
    radioSend(sensor_val_avg);
  }
}
```

(c) Task with Variable Frequency

```
int main( void ) {
  // ...
  xTaskCreate( RadioTask, "Task1", configMINIMAL_STACK_SIZE, NULL,
     mainTASK_PRIORITY, &handleRadio, &radio_knob, 100, 5000, 1);
  xTaskCreate( SensorTask, "Task2", configMINIMAL_STACK_SIZE, NULL,
     mainTASK_PRIORITY, &handleSensor, &sensor_knob, 1, 100, 1);
  vTaskStartScheduler();
}
```

(d) Task Creation and Initialization

Figure 4.13: VaRTOS Task and Application Example

repeatedly increased by a delta defined such that the difference between maximum and minimum knobs is divided by a small number of points. Between each perturbation in k_i, the task is allowed to run for a time period sufficiently long enough to capture active time measurements for tasks with very infrequent activity. This supervisory period is set at $t_{super} = 1$ hour, and hence the mappings are calculated after 4 hours. Many tasks are likely to make heavy use of interrupt subroutines (e.g. for analog to digital conversion, radio transmission, serial communication, etc.). In order for this time to be accounted during the supervisory period, we provide functionality for assigning each subroutine to a particular task using a handle provided during task creation.

Changing each knob value k_i will cause a corresponding change in duty cycle ratio d_i based on the nature of the task. Given minimum and maximum knob values $k_{i,min}$ and $k_{i,max}$ as well as a mapping from k_i to d_i we can construct a utility function $u_i = f(d_i)$ as the convex portion of a logistic function:

$$u_i(d_i) = \frac{2}{1 + e^{-c_i d_i}} - 1, \quad c_i \geq 0, \quad d_{i,min} \leq d_i \leq d_{i,max} \tag{4.5}$$

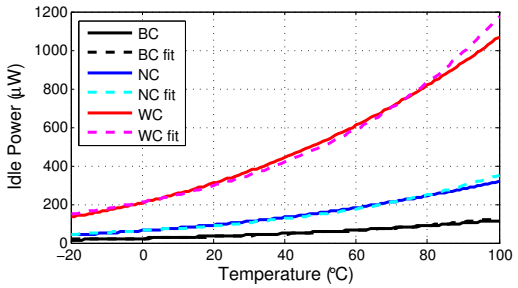
where $d_{i,min}$ and $d_{i,max}$ are task duty cycles corresponding to $k_{i,min}$ and $k_{i,max}$. A logistic function is used because the convex portion of the characteristic s-like curve offers a convenient form for modeling diminishing returns on k_i. Here, c_i governs the convergence rate of u_i from the minimum utility to the maximum utility and is calculated as a function of $k_{i,min}$ and $k_{i,max}$ such that 99% of the utility has been reached by k_{max}. Finally, each utility curve can be arbitrarily increased or decreased by a priority scalar p_i for tasks with intrinsically higher or lower utility than others. This offers a level of customization in addition to specifying $k_{i,min}$ and $k_{i,max}$, allowing the developer to give preference to one task over another.

4.3.2 Online Modeling of Power

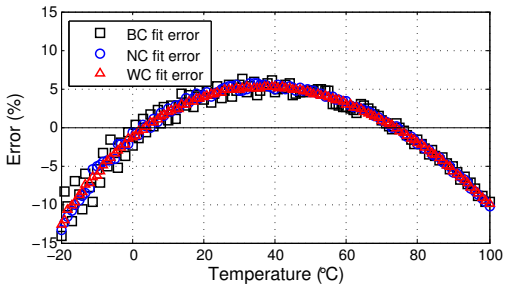
As discussed in Chapter 2, both sleep power and active power are nonlinear functions of temperature. A large portion of this nonlinearity comes from leakage and sub-threshold currents which dominate in P_S. For our TinyOS implementation, we assumed that the temperature to active and sleep power curves were characterized before deployment.

In general, learning these nonlinear curves for each hardware instance could prove difficult with limited resources and without, in many cases, fully fledged math libraries. For example, nonlinear regression is often performed as an optimization problem using a specialized library such as NLopt, requiring more than 300 kB of program space in order to do even rudimentary optimization routines [NLo13] and prohibiting its use in many low power platforms. Knowledge of the closely exponential shape of the sleep power function, however, allows us to linearize the model which in turn allows the use of linear regression to accurately model P_S. Specifically, linear regression is run on $\log(P_S)$, giving offset b_s and slope m_s. The desired sleep power model is likewise computed as $P_S(T) \approx \exp(b_s + m_s T)$. After $P_S(T)$ has been computed, $P_A(T)$ can be modeled by subtracting $P_S(T)$ from active power measurements and continuing with a second linear fit.

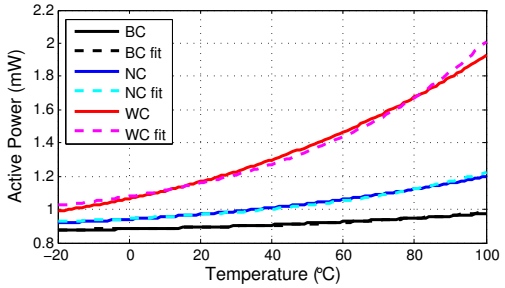
The error between the models described in Chapter 2 and the linear approximation methods described above is shown in Figure 4.14 for three separate power instances representing the best-case (BC), nominal-case (NC), and worst-case (WC) for a 45nm Cortex M3 processor—(a) shows the sleep power model with the corresponding error in (b), and (c) shows the active power model with the corresponding error in (d). For the linear approximation of P_S on the temperature range [-20°C, 100°C], the worst case error is around -15% while on a more temperate range of [0°C, 80°C] the worst case error is around 5%. For most temperature



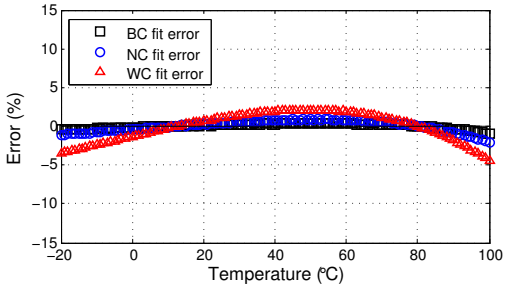
(a) Sleep Power



(b) Sleep Power Error



(c) Active Power



(d) Active Power Error

Figure 4.14: Modeling sleep and active power through linearization.

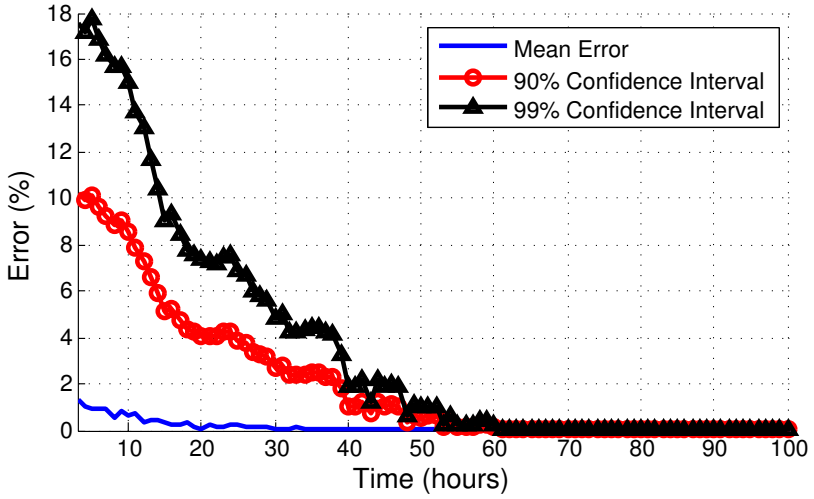


Figure 4.15: Error convergence for sleep power modeling.

profiles this accuracy will be adequate, but deployments in extreme environments can experience the detriments of errors in the linear model of P_S. Because of the added baseline in P_a, the corresponding prediction error is drastically reduced—less than 2% across [-20°C, 100°C] for the best-case and nominal instances and less than 5% for worst-case. These errors can be further reduced using nonlinear regression methods if the computational resources are not a limiting factor; for VaRTOS we have chosen a lightweight design so that resource-constrained low power processors—those that are likely to be used in long lifetime sensing tasks—can easily perform the necessary computations.

Models for both P_S and P_A take some time to converge, before which an accurate prediction for the optimal duty cycle cannot be calculated. Convergence is aided by variations in temperature, giving a variety of points on the $T \rightarrow \{P_S, P_A\}$ curves, and hurt by noise variance in power sensors. For example, if our sensor for P_S takes hourly measurements with additive white Gaussian noise $\sim (0,\ 5\mu\text{W})$, the percentage error of our model has reached a reasonable accuracy after 40 hours and is nearly fully converged after 60 hours. This is shown in Figure 4.15 for 190 different locations within the United States with between 1

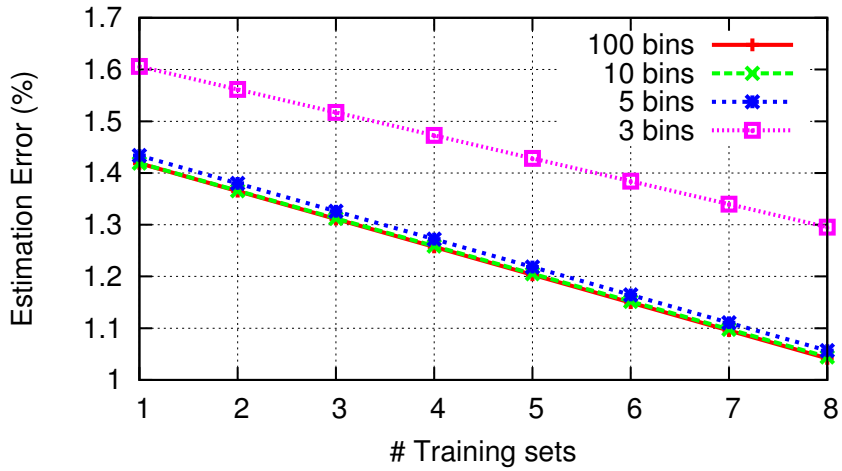


Figure 4.16: Error in average power estimation for varying number of training years and histogram bins.

and 9 years of hourly data in all locations and for processor instances with best-case, nominal-case, and worst-case power consumption.

Determining the optimum duty cycle in Equation 4.2 requires knowledge of temperature distribution across the lifetime of deployment. We were unable to perform online learning of temperature distribution, as it takes the entirety of a year to develop an accurate histogram of the temperature values seen at a given location. We found however that a very coarse representation of the temperature profile suffices for accurate calculations of the optimum duty cycle, and furthermore temperature profiles change very little from year to year for a given location.

Figure 4.16 shows how certain temperature models affect the error in predicting average power consumption for P_S across the lifetime of the system (in this case, 1 year). The x-axis here represents the number of years of temperature data used to train the model before testing on a single year. Each line represents a certain number of bins used in a histogram representing f_T for a given location. This figure shows that the decreasing estimation error indicates that temperature profiles change very little from year to year, and because of this using multiple years to build f_T only serves to decrease the prediction error in years to come; and

while a 3-bin histogram is inadequate to fully represent the temperature profile for a given location, there is very little benefit in representing f_T with more bins than 5 and even less so with more than 10. Because of this, for a given location we train with as many previous years as are available and we use a 10-bin histogram to represent f_T.

4.3.3 Maximizing Application Utility

Given temperature to sleep and active power mappings, temperature profile, and a lifetime requirement, the optimum system-wide duty cycle is given under practical conditions by Equation 4.2 in Section 4.1.2. Given the system-wide duty cycle, we now seek a distribution of duty cycle to multiple tasks that maximizes system-wide utility. Because we have defined utility u_i to be a logistic function, we can use a greedy approach when optimizing utility. The optimization routine in VaRTOS is a two step process: (1) attempt to assign the minimum duty cycle needed for each task in order of decreasing priority, and (2) continue distributing computational time in small increments to those tasks yielding the largest marginal utility until no active computational time is left. Duty cycle is incrementally added by a small fraction δ to those tasks with the largest marginal utility until the system-wide duty cycle is reached. Task knobs are then assigned according to the knob to duty cycle mapping for each task.

Figure 4.17 illustrates the optimization process in VaRTOS. To begin, the system is initialized with task creations, energy and lifetime specifications, and a location-specific temperature model. If at least one task has been created, the scheduler begins operation and we enter a model convergence state. While in this state, hourly temperature and power measurements are collected and knob values are incremented to construct to construct the knob to duty cycle map for each task. The optimization routine cannot complete until both models have converged, after which linear regression and linearization are used to fit the knob

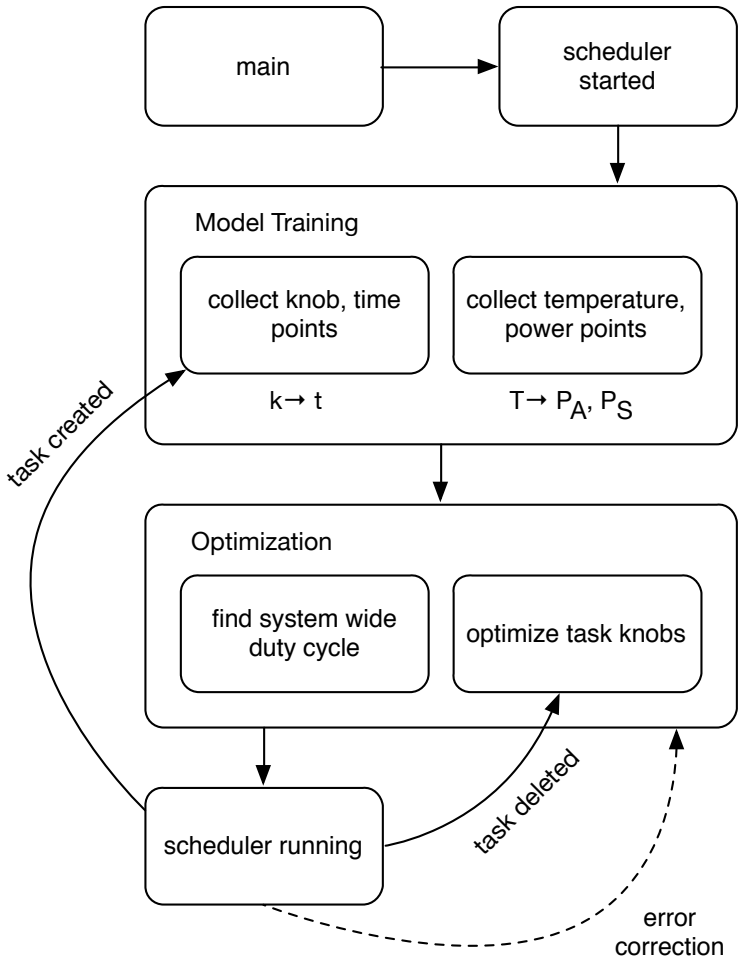


Figure 4.17: VaRTOS state chart

to duty cycle and temperature to power curves, respectively.

In the optimization state, the various task duty cycles are calculated and the corresponding knob values are assigned to the appropriate tasks. At this point we begin steady state operation in the 'Scheduler Running' state. Potential reasons for leaving this state include task creation (necessitating learning the new task's knob to duty cycle mapping and re-optimizing) or task deletion (requiring only re-optimization). Because the modeling tasks only run on an as-needed basis, these are implemented as OS tasks with null-valued knobs. This allows for easy suspension and resumption of these tasks as necessary.

The dashed line on Figure 4.17 represents an optional feedback error-checking mechanism that can help for online readjustment of poor initial power model construction (e.g. for cases where measurement of P_S and P_A is particularly noisy). This can be done by comparing true energy expenditure with predicted expenditure, if such a sensor exists, and using the error to apply proportional feedback. We do not explore results for a feedback mechanism in this work.

4.3.4 Evaluation

We evaluate VaRTOS different hardware instances and deployment scenarios (temperature profiles) across a lifetime of 1 year. Because it would be impractical to physically deploy these applications, we rely on VarEMU for the evaluation. When starting VarEMU, we provide a configuration file with parameters for the power model described in Section 3.1.4. We evaluate the system with three instances (nominal, best-case, and worst-case). We also provide a trace of temperature based on hourly temperature data from the National Climactic Data Center [US14] for three locations: Mauna Loa, HI ('best-case': mild temperature, very little variation), Sioux Falls, SD ('nominal-case': average temperature and variation), and Death Valley, CA ('worst-case': extreme temperature and variation). For every hour elapsed on the Virtual Machine, VarEMU reads a new line from the temperature trace file and changes the temperature parameter in the power model accordingly. In order to accelerate the simulation (which would otherwise run in real-time), we use a time scale of 1:3600, resulting in a total simulation time of approximately 2.5 hours for a lifetime of one year.

4.3.4.1 Energy consumption

In order to achieve accurate energy consumption to meet a lifetime goal, VaRTOS needs to accurately be able to achieve the overall system duty cycle. To test this,

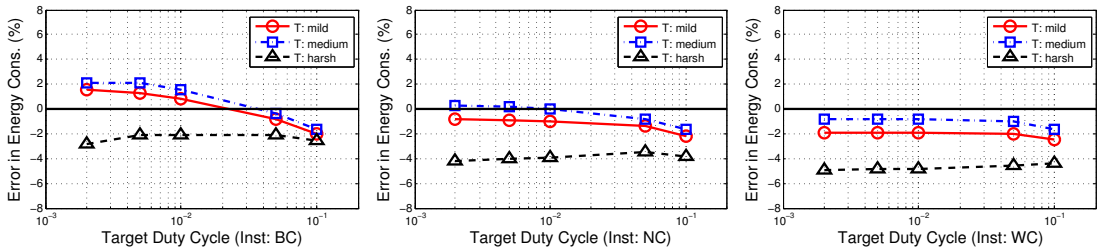


Figure 4.18: Error in energy consumption for various optimal duty cycles across deployment scenarios.

we constructed a simple application with only a single task containing a knob with fine granularity values. We then specified various values for E and L that would ideally lead to a particular system-wide duty cycle for each of the power instance models (best-, nominal-, and worst-case) as well as three temperature profile instances (harsh, medium, and mild). The target duty cycles ranged from 0.2% to 10%, and the resulting errors in energy consumption are shown in Figure 4.18. Errors are larger in harsher environments, where any errors in the power models will be magnified. In the worst case, an error of 4.9% in energy consumption is seen for a harsh environment and for the worst-case power instance (far right plot in Figure 4.18). This means that, in the worst case, a 5% guard band in lifetime or in energy is necessary if the lifetime goal is to be treated as a hard constraint.

4.3.4.2 System Utility

In this section, we comparing the resulting utility of a single task app running in VaRTOS to that of an all-knowledgeable oracle system. Unlike the VaRTOS system, the oracle system has (1) complete knowledge of the temperature profile for the test year; (2) perfect knowledge of task behavior (i.e. mapping from knob to time); (3) full accuracy models for P_S and P_A; and (4) zero overhead for optimization routines. Figure 4.19 shows the utilities for both the oracle and VaRTOS. In most cases VaRTOS achieves within 10% of the oracle utility, and is

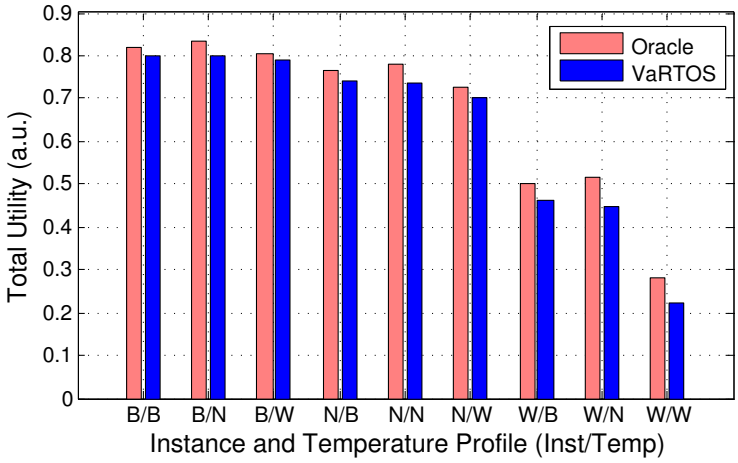


Figure 4.19: Total utility for VaRTOS vs. oracle system

as much as 20% off in the worst case.

4.3.4.3 Runtime Overheads

Energy consumption by the various VaRTOS subsystems must be minimized in order to prevent energy wastage. Similarly, the memory required for VaRTOS must be kept reasonably low in order to make it a viable option for resource-constrained platforms.

Memory overhead of the VaRTOS system is application and configuration dependent. As a baseline, VaRTOS requires an increase in code memory usage over the vanilla FreeRTOS framework from 2.29 kB to 6.80 kB (a 4.51 kB increase). This includes a lightweight library for math functions required for optimization routines (including exponential, logarithmic, and square root functions) as well as a preemptive scheduler. If a full math library needs to be used for the application itself, these functions can be replaced and the overhead amortized. In terms of data memory, an additional 508 bytes baseline is required (480 bytes of this is due to the power learning procedure and, if the developer is so motivated, can be reused after the models have converged). An additional 46 bytes per task is

also required for knob modeling and other parameters. Finally, the temperature profile is stored in program ROM as a constant array and uses 10 bytes.

The largest energy overhead in VaRTOS comes from the scheduler itself, which, if context swaps occur every 10 ms, causes a baseline system duty cycle of 0.1%. This ratio can be decreased if coarser granularity context swaps are acceptable. The power consumption attributed to this 0.1% depends on the power consumption of the processor and the environmental temperature, but even in the worst case the scheduler adds only marginal energy consumption on top of the baseline sleep power.

Other potential energy consuming processes attributed to VaRTOS include knob modeling, power measurement and fitting, finding the optimal d^*_{sys}, and finding the optimal knob values. The amount of processing time spent in these tasks is negligible: reading power and temperature takes 250 μs and occurs only 40 times over the course of a deployment (10 ms total); knob perturbations take 48 μ s and occur $4{\cdot}N$ times (for N tasks); performing the linear regression for power curves and for determining task knob to time mappings takes 40 ms and occurs twice (P_s and P_a) per deployment and once per task; finding system-wide duty cycle takes 54 ms and occurs once unless tasks are deleted and created after the initial optimization; and finally finding optimal per-task duty cycle and knob values takes 345 μs. In total, these added tasks consume less than 1 mJ in the worst-case for a 1 year deployment, a negligible overhead if our energy budget is 12960 Joules (2 AAA batteries).

4.4 Application Results

Higher duty cycles allow the sensors to stay "on" for a longer time and capture more data during deployment. This typically increases accuracy and shortens response times in high fidelity real-time sensing tasks such as object localization

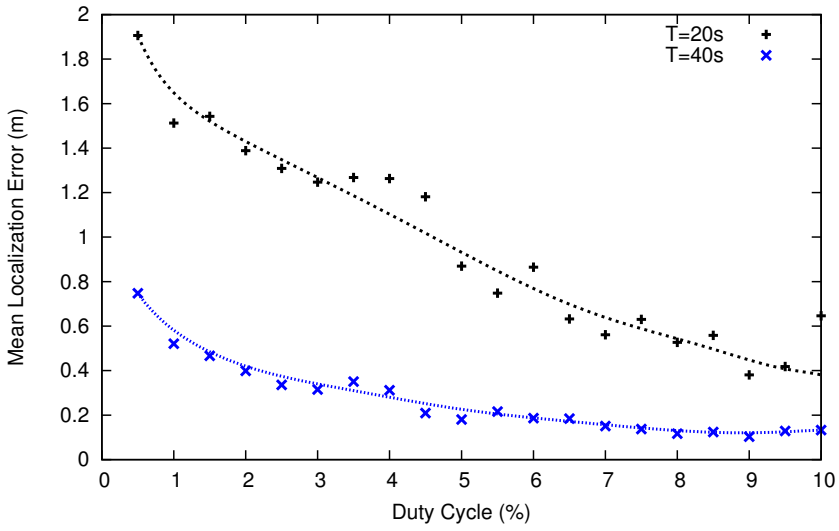


Figure 4.20: Mean localization error across duty cycles.

and tracking [ZSB10]. For instance, Figure 4.20 quantifies the effect of different duty cycles on the accuracy of sound source localization with a network of 20 acoustic (e.g. microphone) sensor arrays using Maximum Likelihood Estimation (MLE). The simulations were adapted from [ZSB10] and represent 200s of application execution in real-time. Each point in the plot is generated from an average of 200 simulation runs with varying sensor on/off schedules. It shows that with increasing duty cycles (as well as with longer data acquisition times), the performance of the application improves as the mean localization error decreases. The application estimates the location of a target based on line-of-bearing (LoB) measurements from the sensor arrays deployed randomly in a 10m $\times$ 10m field. Each sensor requires approximately 1 second on an embedded ARM-based processor to compute one LoB measurement from raw audio samples. The error in sensor measurements is assumed to be less than 10%.

Moreover, Figure 4.20 demonstrates that the error also decreases with time, for a specific DC, as the algorithm waits to collect measurements from all the sensors before fusing them to obtain the location estimates. This result is important when the sensor nodes follow asynchronous wake-up and sleep schedules to avoid the

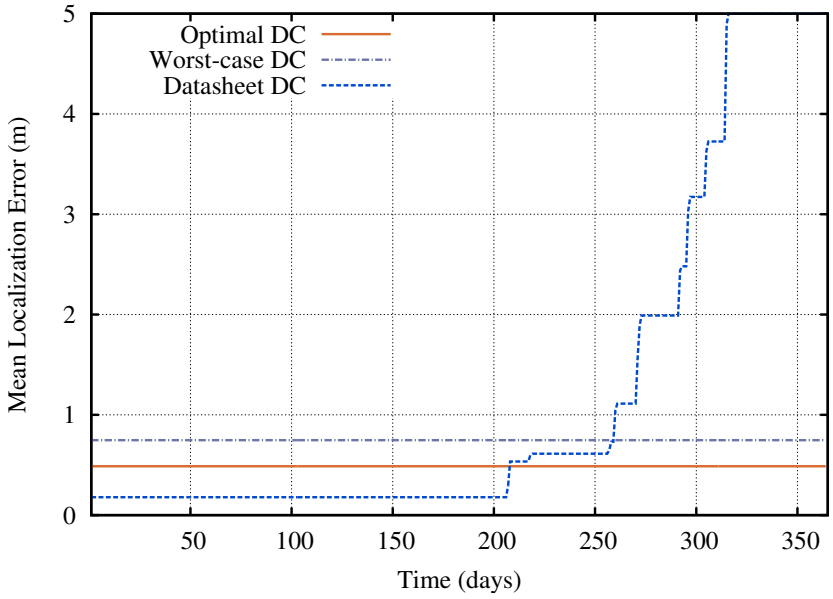


Figure 4.21: Mean localization with variability-aware, worst-case and datasheet-based duty cycle schedules.

control message overhead associated with synchronous duty cycling. For instance, the sensors duty cycle asynchronously with a period of 200s in our simulations of the localization application. However, this is based on the implicit assumption that the target remains static during data collection to enable the sensors to obtain relevant LoB measurements. Correspondingly, in Figure 4.20, the target is assumed to be static for a period of 20s and 40s for the respective plots. This constraint on target motion is due to the limited number of nodes in the simulation scenario, and the low duty cycles in question.

In accordance with these results, Figure 4.21 demonstrates that the algorithm generates estimates of target location with a lower mean error in presence of variability-aware optimal schemes as compared to worst-case schedules for the same battery capacity and lifetime constraints, using the evaluation scenario described in section 4.2.1.1. Although the mean error is the lowest at the start of the deployment with DC schedules determined from datasheet values, it increases steeply towards latter half of the deployment timeline as sensors exhaust their

batteries before the intended lifetime of 1 year. In all these simulations, it is assumed that targets appear at the center of the field once every 200s and remain static for a period of 40s at each appearance. As with the previous result, this constraint on target motion could be relaxed if a larger number of nodes were available, or if more energy was available to each node (larger batteries). In the later case, as indicated in Figure 4.11, the benefits of the variability-aware scheme over the worst-case scheme would be smaller.

CHAPTER 5

Variability-Aware Algorithmic Choice

A software stack that changes its functions to exploit and adapt to runtime variations can operate the hardware platform close to its operational limits. The key to this achievement is to understand the information exchanges that need to take place, and the design choices that exist for the responses to variability, in terms of what the character of the responses is, where they occur (i.e., which layer of software), and when they occur (design or run time).

The range of possible responses that the software can make is rich: control the rate of computation through control of task activations; use a different set of hardware resources (e.g. minimize use of a power hungry module); change the algorithm or algorithm parameters (e.g., switch to an algorithm that yields lower quality, but requires fewer resources); dynamically recompile code to better match a platform's capabilities; and change hardware's operational setting (e.g., tune software-controllable control knobs such as voltage/frequency).

These changes can occur at different places in the software stack or at runtime with corresponding trade-offs in agility, flexibility, and co-existence: explicitly coded by the application developer making use of special language features and API frameworks (*application-driven*), or transparently managed by the operating system resource scheduler (*system-driven*).

A seen in section 1.2, a large body of work exists in system-driven management of software and hardware parameters, as well as dynamic recompilation and hardware resource management. In chapter 4 we presented prototype system soft-

ware stacks that adapt to variations by controlling task activation and adjusting duty cycle, i.e. the rate of activity of the system, based on lifetime requirements, energy availability, and application parameters. Task activation control is a valuable adaptation strategy for embedded sensing systems, where time spent in sleep mode may account for most of the energy dissipated by the system across its lifetime. For systems with significant variation in active power consumption, a *choice* of software to be executed provides further opportunities for optimization. Algorithmic choice, however, is typically driven by applications, and not the runtime system.

In this chapter we introduce the concept of ViRUS: Virtual function Replacement Under Stress. ViRUS is loosely related to polymorphic engines [Yet93] in that it is used to transform sections of a program into different versions with alternate code paths that perform roughly the same functionality. Polymorphic engines are used to intercept and modify code transparently, typically for malicious purposes such as hiding malware functionality from anti-virus software. In ViRUS, the different code paths provide varying quality-of-service for different energy costs. Mutations from one version to another are triggered by vectors of variability and energy stress. ViRUS can thus be seen as a dynamic or flexible version of delegation [Lie86] for object-oriented systems, where lookup rules are used to bind methods to objects. A specific block of code may be activated in ViRUS, for example, when processor temperature reaches a certain threshold. A second block may be activated when remaining battery capacity drops bellow a specified percentage. The different code blocks may be either standard library functions provided by the runtime system or alternative implementations provided by application programmers. Per-application configuration files determine when and under what circumstances code mutations should be triggered. The runtime system monitors the energy stress vector and transparently triggers mutations at appropriate times.

We present a realization of ViRUS in the form of a framework for transparent function replacement in shared libraries demonstrated with a polymorphic version of the standard C math library in Linux. Microbenchmarks show that both systems can monitor hardware and environmental conditions and trigger code mutations with negligible overhead. We show how ViRUS can help users in developing and analyzing tradeoffs between accuracy and energy consumption in different contexts and reduce the energy consumption with user-defined quality-of-service degradation.

5.1 System-Driven Algorithmic Choice

Operating systems implicitly provide multiple code paths through their Hardware Abstraction Layer. A specific code path from applications to hardware will be determined according to hardware availability and configuration. For instance, in embedded systems where multiple communication channels are available (e.g., a mobile phone with Wi-Fi and a 3G data connection), an application may open a communication socket, which can be bound to a specific interface according to application requirements or automatically, e.g., through a priority mechanism. While applications may be given adaptation choices (e.g. using the `setsockopt` system call to bind a raw socket to a specific interface), a quality vs. resource usage tradeoff is typically either not present or implicit (e.g. connections may be faster with Wi-Fi than with 3G or vice-versa).

One example of a system service with elastic quality is found in the Android operating system location service, which provides applications with geographic location information [Goo]. In a typical mobile phone, location information can be derived from a series of sensors (e.g. GPS, assisted GPS, Wi-Fi triangulation) with different energy cost and location accuracy. For each of these sensors, location accuracy may also be variable according to ambient conditions (e.g. availability

of satellites and nearby wireless access points). If the application needs only "best effort" location information, it simply acquires a reference to the system's location service, and requests notifications based on change of location. The system manages location sensors and quality autonomously. Applications that need fine grain location information may request a location provider that meets a given accuracy criteria. Applications may also poll the system or request notifications for changes in location accuracy.

We propose ViRUS as an application runtime support system where the operating system adjusts service quality according to variability-aware policies. To accomplish this, we (i) leverage techniques for algorithmic choice, building this capability into the application support system; (ii) allow for adaptation with minimal application intervention, as in the example of the Android location stack; (iii) expose service quality information to the applications, so that developers can constrain or guide adaptation according to application requirements; (iv) build variability-aware adaptation policies that expose variability information to the various layers of the software stack and drive system adaptation.

5.2 System Design for Algorithmic Choice

Several services provided by the operating and runtime support are amenable to adaptation and may be extended to support elastic quality levels. These services include numeric and signal processing services (e.g. with variable numeric precision), multimedia services (e.g. with variable video encoding and decoding quality), sensing stack (e.g. the aforementioned location service), and communication stack (e.g. diversity of communication channels). Different versions of each of these services can be provided to applications through a common interface, or through a wrapper that adapts versions with different interfaces.

Assuming that, for a given service, multiple libraries with identical interfaces

and semantics are available, a simple strategy for providing different versions of a service would be to link an application with a shared library selected from a poll of available alternatives when the application is loaded. While this would require minimal extensions to the OS and only incur a small overhead when the application is loaded, it would also allow only for coarse-grained adaptation, as adaptation decisions are only made when the application is loaded. Because operating conditions may change with time, long running applications may be bound to sub-optimal choices of service level.

To allow for dynamic adaptation, the dynamic linking process could be altered to allow changes in the Procedure Linkage Table (PLT) of an application after it is populated. The PLT stores references to shared library functions used by the application. The dynamic linker populates the PLT on demand, and function addresses are not bound to processes until the functions are called. If multiple versions of a function were present, the linker could choose a target implementation based on application parameters and the system's variability aware policies. A signal from the operating system to the application could, for example, invalidate entries in the PLT, triggering a change in choice of code paths.

To allow for transparent application adaptation without changes to the dynamic linking process, in ViRUS we build a thin wrapper around libraries to provide variability-aware algorithmic choice. This wrapper takes adaptation configuration parameters from applications (e.g. what functions are sensitive to sources of stress and amenable to adaptation, bounds for quality requirements), and handles variability messages from the operating system that trigger function replacement. The wrapper also adapts multiple libraries with common functionality but different function signatures into a common interface.

5.3 Library Generation

Given multiple implementations $f_0(\cdots), f_1(\cdots), \cdots, f_n(\cdots)$ of a function featuring the same signature, ViRUS exposes a single function $f(\cdots)$ to applications. In our C language implementation, this is accomplished by declaring $f(\cdots)$ as a function pointer, and dynamically assigning the pointer to the address one of the $f_n(\cdots)$ implementations. Because calls to functions and function pointers are identical in C, there is no indirection between a call to the function entry point f and the function call $f_n(\cdots)$. In other words, the function pointer $f(\cdots)$ is aliased to one of the implementations; for example, if the function pointer $f(\cdots)$ is assigned to $f_1(\cdots)$, an application call to $f(\cdots)$ translates to the same sequence of operations as a call to $f_1(\cdots)$.

In general, multiple implementations of a function f may have different type signatures. For every f_n function implementation we therefore must create a wrapper function that matches the signature of pointer f. This process is automated in ViRUS as illustrated by Figure 5.1. The `F_ONE` macro exposes function `f` of type `rettype` and one parameter of type `argtype` as function `_var_name_level`, where level is an integer that defines an order of quality between the multiple implementations. Lower numbers represent higher quality. In this example, multiple implementations of the exponential function are declared with different signatures (5.1.a). A function declaration macro (5.1.b) uses the double pre-processor directive (`##`) to concatenate function name, quality level, return type, parameter type, and function to be called into an inline function definition. In the example, the macro maps a function with one parameter into a wrapper with one parameter. Similar templates are used for other combinations of parameter mappings. The wrapper templates are used as in 5.1.c, and the resulting code after pre-processing is shown in 5.1.d.

Taking a design reference from [BC10] and [LMM07], we expose the multiple

```
double exp (double x);
float expf (float x);
static inline float fastexp (float p) { ... }
static inline float fasterexp (float p) { ... }
```

(a) Function Signatures

```
#define F_ONE_ONE( name, level, rettype, argtype, f ) \
rettype __var__ ##name ##level (argtype arg) { return f(arg); }
```

(b) Wrapper function prototype

```
F_ONE_ONE(exp, 0, double, double, exp)
F_ONE_ONE(exp, 1, double, double, expf)
F_ONE_ONE(exp, 2, double, double, fastexp)
F_ONE_ONE(exp, 3, double, double, fasterexp)
```

(c) Usage example

```
double __var__exp0 (double arg) { return  exp(arg); }
double __var__exp1 (double arg) { return  expf(arg); }
double __var__exp2 (double arg) { return  fastexp(arg); }
double __var__exp3 (double arg) { return  fasterexp(arg); }
```

(d) Resulting code

Figure 5.1: Exposing multiple functions with mismatched signatures.

```
#define F_MUTATORS( name, max_level) \
int __v_ ##name ## _curr_level = 0; \
void v_ ##name ## _set_qlevel(int lvl) \
{ \
    if (lvl < 0) { \
        lvl = 0; \
    }; \
    if (lvl > max_level) { \
        lvl = max_level; \
    }; \
    __v_ ##name ## _curr_level = lvl; \
    v_ ##name =__v_##name[lvl]; \
} \
int v_ ##name ## _get_qlevel(void) { return __v_ ##name ##
   _curr_level;} \
int v_ ##name ## _avl_qlevel(void) { return max_level; }
```

Figure 5.2: Mutator function templates.

implementations of a function as a function array ordered by quality. For each f, a four-function API is exposed: f itself, two mutator methods for quality level (getter/setter), and a method that returns the number of implementations available. The generation of these methods is automated with a set of macros that take the function name and total number of implementations as parameters, as illustrated in Figure 5.2.

For every function to be exposed, ViRUS requires of a function pointer; a series of alternate function declarations (as in Figure 5.1.c); function mutator methods automatically generated with a call to the `F_MUTATORS` macro, e.g., `MUTATORS(exp, 3)`; a function array with pointers to each of the multiple wrappers; and a constructor method. Because the function array is of arbitrary size, it cannot be automatically generated with pre-processor macros. This array

```
#define LIB_INIT( name, strname ) \
void __attribute__ ((constructor)) __var__ ##name ## _init(void); \
void __var__ ##name ## _init(void) { \
  __v_ ##name ## _curr_level = 0; \
  v_ ##name =__v_##name[0]; \
  v_register_knob(strname, v_ ##name ## _set_qlevel, v_ ##name ##
     _get_qlevel, v_ ##name ## _avl_qlevel); \
}
```

Figure 5.3: Constructor template. When the library is loaded prior to reaching the application's main function, the constructor for each function is executed.

could be automatically generated, for example, using metaprogramming templates in C++. In our current implementation, the library programmer must create this array manually.

For each function, a constructor method is automatically generated and executed to set the default function pointer and quality level when the library is first loaded (using the constructor function attribute syntax of GCC). Figure 5.3 illustrates constructor implementation. This method also registers the name of the function along with its mutator methods with the ViRUS controller that handles messages from the operating system and triggers function mutation according to application and system configuration.

5.4 ViRUS controller

The ViRUS controller monitors hardware and operating environment for vectors of stress and dynamically triggers function mutations according to system and application configuration. Each application using a ViRUS library has its own controller featuring all control knobs (function mutators), function replacement rules, and sensor monitors for its stress vectors.

Each function in a ViRUS library is associated with a constructor, as illustrated in Figure 5.3. Each constructor registers the function with the ViRUS controller using a name and a set of mutators. The collection of these register entries forms the knob table – i.e., the set of possible mutation actions – in the ViRUS controller for an application.

Each application is associated with a configuration file listing its function replacement rules. Each rule in the configuration links a function with a stress sensitivity vector and a set of acceptable quality levels. A sensitivity vector is defined as a range of values for a sensor, for example, temperature between 0 and 40°C or instant active power between 100 and 500mW. The set of acceptable quality levels is defined as an integer range from the highest to lowest quality acceptable for each function while operating under each sensitivity vector.

Table 5.1 shows a ViRUS configuration table for a hypothetical application. Each rule describes a function, priority, sensor, range for the sensor, and set of acceptable quality levels. The application is sensitive to battery level and temperature. Function f may operate in quality levels 0 or 1 when temperature is between 0 and 40°C, and quality levels 2 or 3 when temperature is between 40 and 100°C. Lower numbers represent higher quality levels. When multiple sensors may trigger mutations for the same function, rules are resolved in order of priority, with lower numbers representing higher priority. Function g may operate in quality levels 0, 1, or 2 when remaining battery is between 20 and 100%, but only in quality level 2 when the battery level is bellow 20%. A lower priority rule for temperature further refines the choices to level 0 when temperature is between 0 and 60°C, and levels 1 and 2 when temperature is between 60 and 100°C. Due to different priorities, function g will operate at quality level 2 when battery is bellow 20% even if temperature is bellow 60°C. When rules are defined for a function with different sensors and the same priority level, the most energy conservative rule (i.e., the one with the lowest quality set of alternatives) is interpreted as having

function	priority	sensor	range	quality
f	0	temperature	$[0, 40)$	$\{0, 1\}$
f	0	temperature	$[40, 100)$	$\{2, 3\}$
g	0	battery	$[20, 100)$	$\{0, 1, 2\}$
g	0	battery	$[0, 20)$	$\{2\}$
g	1	temperature	$[0, 60)$	$\{0\}$
g	1	temperature	$[60, 100)$	$\{1, 2\}$

Table 5.1: ViRUS configuration rules example. Each rule associates a function with a vector of sensitivity (sensor and range) and acceptable quality levels. Rules are resolved in order of priority.

the highest priority. Likewise, when multiple quality choices are available for a function after resolving all the rules, the lowest quality version is chosen. A special function name, $\star$, is used for rules that apply to all functions. If an application configuration mixes function specific and wildcard rules, function-specific rules override wildcard rules.

Stress sensors are exported to the ViRUS controller by a variability monitor (described in Section 5.5) as a table of sensor names and a floating point variable to access current sensor values. The ViRUS controller is implemented as a library that is linked with each application. A constructor method parses the rules file for the application. Configuration files are stored under `/etc/virus/` and named identically to the application executable. File name is obtained with the `realpath` standard library function. Each line in the file reflects a line in Table 5.1. For each line that is parsed, we iterate through the tables of knobs and sensors searching for function and sensor names that match the rule. If matches are found, a rule with the knob, sensor, range, and quality is added to the rules table. If there are no matches for the function or sensor name, the rule is ignored. If no configuration file is found for the application, a default global configuration

file containing wildcard rules is used. Linking order ensures that the function constructors are executed before the ViRUS controller constructor, and therefore knobs are registered before rules are parsed.

Mutations are triggered by a message from the variability monitor. These can be of a periodic nature or range-based alarms for sensors, according to application configuration. The controller library constructor configures variability alarms and installs a message handler. When a message is received, we iterate through all the rules to find replacements for each function using the algorithm in Figure 5.4. For every unique knob name, we find the corresponding set of rules, starting with the wildcard ($\star$) rules. For each of the rules for a knob name, we check current sensor values against the range for the rule. If the sensor value is within range, we mark the rule as matched. Starting with a set of all possible quality levels, we iterate through matched rules in order of priority to refine the set of allowable quality levels. Finally we set the quality of the function associated with the rule (or all functions in the case of a wildcard rule) to the maximum of the set of allowable qualities after processing the rules. If this quality is a higher number than the number of alternate implementations provided the function will be set to the lowest quality (highest number) available.

5.5 Variability Monitor

ViRUS monitors stress vectors for variability and notifies applications to trigger function replacements. The monitor is built with two components: a system daemon that monitors sensors and triggers alarms upon certain conditions, and a small library linked with applications that registers processes to receive adaptation triggers and handles notifications from the daemon.

Figure 5.5 illustrates the interaction between applications and the monitor daemon. The monitor listens for connection on a UNIX-domain socket. A constructor

```
Data: Rules: set of (knob, priority, sensor, range, quality) tuples
RulesToProcess ← Rules;
KnobName ← ⋆ ;
while RulesToProcess ≠ ∅ do
    RulesForKnob ← {r ∈ RulesToProcess : r.knob.name = KnobName};
    MatchedRules ← ∅;
    forall the rule ∈ RulesForKnob do
        sensor ← rule.sensor.value();
        rmin ← rule.range.min;
        rmax ← rule.range.max;
        if rmin ≤ sensor < rmax then
            MatchedRules ← MatchedRules ∪ rule;
        end
    end
    QualitySet ← {r.quality ∀r ∈ MatchedRules } ;
    forall the rule ∈ MatchedRules ordered by rule.priority do
        if (QualitySet ∩ rule.quality) ≠ ∅ then
            QualitySet ← QualitySet ∩ rule.quality;
        end
    end
    Set quality of functions matching KnobName to max(QualitySet) ;
    RulesToProcess ← RulesToProcess \ RulesForKnob;
    KnobName ← rule.knob.name for some rule ∈ RulesToProcess ;
end
```

Figure 5.4: Function replacement algorithm

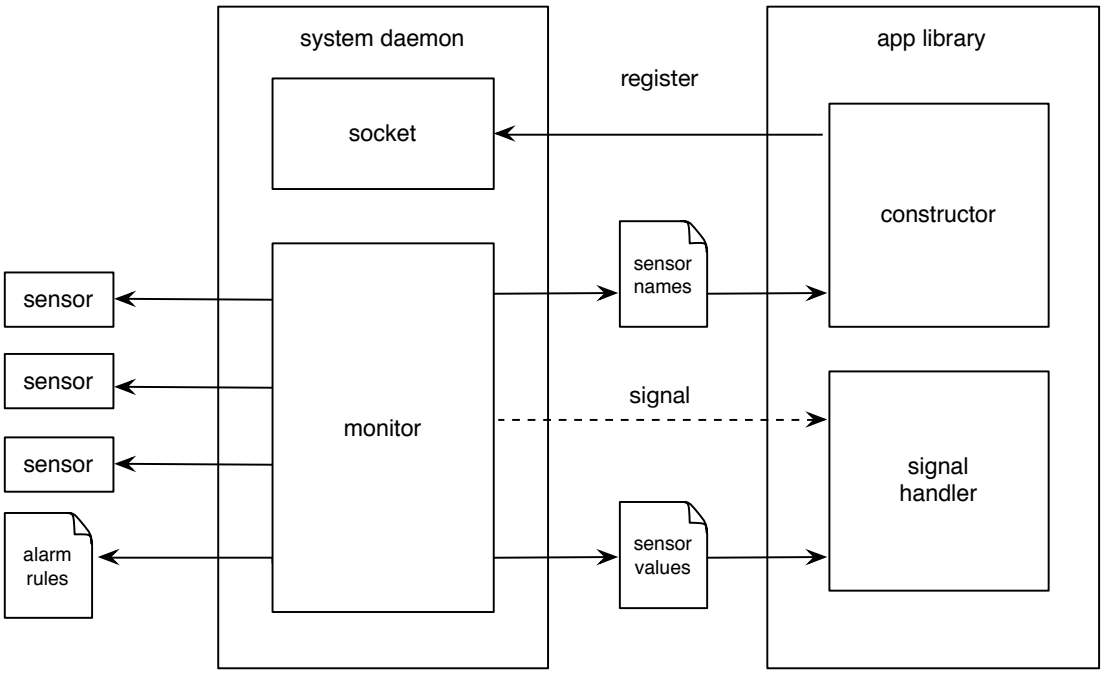


Figure 5.5: Variability Monitor Architecture

method registers the application process identifier with the monitor through this socket upon library initialization. This constructor also reads a sensor descriptor file to discover available sensors and build the sensor table used when matching application function replacement rules, and registers a signal handler for the application. The monitor daemon periodically reads sensors and writes their values to a file. A sensor configuration file for the monitor determines sampling rates and rules for generating alarms. These rules may determine that alarms should be raised, for example, when a sensor value exceeds a certain threshold or when sensor values change more than a certain percentage across subsequent samples. When sensor values lead to an alarm, the monitor daemon goes through a list of registered processes and sends each of them a signal. The signal handler in the application reads sensor values from the monitor and triggers function replacement according to the algorithm in Figure 5.4.

The application library interacts with the monitor through a socket (to register the application), files (to discover available sensors and to read sensor values), and

signals. The constructor method for the monitor library starts by connecting to a UNIX domain socket provided by the monitor server, and sending the application process ID (obtained with the `getpid` standard library function). It then opens the sensor descriptor file written by the server. This file contains a single line with sensor names separated by spaces, and the sensor name tokens are used to construct the sensor table for the ViRUS library. Sensor names are linked to function replacement rules when the application configuration file is parsed. Finally, the constructor method registers a handler for the `USR1` signal.

When the process receives the `USR1` signal, the application must read current sensor values to determine if function mutations should be triggered. Because the signal itself does not convey information other than signal number, sensor values are communicated from the monitor daemon to the application through a shared file. This file contains the last sample for each sensor in the same order as they appear in the sensor descriptor file. Values are parsed and the sensor table for the application is updated with the latest samples. When rules are matched (as per Figure 5.4), reading a sensor value corresponds to reading a floating point variable in memory.

In the variability monitor daemon, hardware and software sensors are abstracted through drivers that expose the sensor name and current value through a floating point function. Available sensors depend on the underlying hardware platform, and are linked with the monitor system service at compile time. In the future this could be extended to allow for dynamic discovery of sensors through a plug-and-play driver architecture. Our VarEMU build of ViRUS includes sensors for temperature, frequency, voltage, and power as determined by the power model in the emulator. Our standard PC ViRUS build exposes frequency and average CPU utilization as a proxy for energy consumption.

For each sensor, a user-defined configuration file determines sampling rate in Hz and alarm rules. Four types of alarm are defined: value greater than, value

equal to, value smaller than, and change in magnitude. The change in magnitude rule will raise an alarm if the sensor has changed by the expressed percentage since the last sample. The four basic alarm types can be dynamically changed without requiring re-compilation, but more complex alarm rules must be attached to sensors through a software module in compile time.

After acquiring a sample, the monitor iterates though sensor rules to search for a match. If a match is found, a signal is sent to all processes that registered their PIDs. If a signal can't be sent to a process, that process is assumed to have finished, and the PID is removed from the list of registered processes. While variability alarms are global, function mutation rules are defined on a per-application basis and therefore a signal may not lead to a mutation.

5.6 Evaluation

For the results in this section, we built our ViRUS libraries and applications for the ARM9 architecture and ran it on our prototype Linux system for VarEMU described in Section 3.2. Unless specified, energy results are normalized across the runs under comparison, and obtained from a nominal (typical) instance running under nominal voltage and temperature. Energy numbers are obtained through the Linux system call interface for VarEMU as described in Section 3.2.2. We use the `-O3` compiler optimization flag for all tests.

5.6.1 Memory Usage

The ViRUS controller is implemented as a shared library that handles: 1) knob registration for multi-quality functions; 2) parsing of application-dependent mutation rules; 3) registration and handling of signals from the variability monitor and reading of sensor data; and 4) function replacement. Combined, these functions use a total of approximately 2.8KB of code memory. When standard library

functions needed to perform operations in the ViRUS controller (e.g., `strcmp`, `connect`) are statically linked with the library in an application, code memory usage totals approximately 8KB.

Adaptation knobs (i.e., available functions) and replacement rules are defined on a per-application basis, and so tables of available knobs, rules, and sensors must be kept for each process. We chose to define the size of these tables statically, avoiding the overhead of dynamic memory allocation when rules and knobs are parsed and registered. Each entry in the knob table contains a string name, and pointers to each of the three mutator functions (getter, setter, and maximum for quality). Sensor entries have a name and floating point value. Finally, rules have a pointer to a knob, a pointer to a sensor, an integer priority, integer minimum and maximum quality, and floating point sensor range (minimum and maximum). Sensor and knob names have a maximum length of 16 characters. For a 32-bit ARM architecture, each entry in the knob, rule, and sensor tables occupies 32, 32, and 24 bytes, respectively (some space is wasted due to memory alignment directives). We define a default number of entries of 16 for the sensor table, and 64 for the knob and rules table. Combined with other internal variables in the controller library, this results in approximately 7KB of data memory being used for each application that links with the ViRUS controller library.

Code memory used by the multiple versions of a function depends on the nature of the function and the implementations. Section 5.6.3 presents a library for variable quality math functions that adds approximately 10% memory overhead to a single high-quality implementation. The function wrappers and mutators required for each ViRUS-compliant function add negligible overhead for each function: < 100 bytes in code and $4(n + 1)$ bytes in data memory, where n is the number of quality levels.

Operation	Dependency	KCycles		
		$N = 1$	$N = 5$	$N = 10$
Process registration	—	161.5		
Knob registration	# functions	15	16.5	18
Sensor discovery	—	232.7		
Rule parsing	—	238.0		
Sensor reading	—	54.4		
Function replacement	# rules	1.6	6.6	13.1
`exp` (reference)	—	3.5		

Table 5.2: ViRUS runtime overheads

5.6.2 Microbenchmarks

In this section we evaluate the runtime overheads of ViRUS through a series of microbenchmarks. These overheads include knob registration, sensor discovery, rules parsing and initialization, process registration with the variability monitor, signal handling, and function mutation.

Table 5.2 shows the number of cycles taken by the various ViRUS operations. For operations that have $\mathcal{O}(N)$ complexity (where N may be the number of knobs or replacement rules), we show the number of cycles taken for $N = 1$, 5, and 10. The $\mathcal{O}(N)$ dependency of operations that involve file I/O (rules parsing and sensor discovery and reading) is negligible compared to the baseline runtime of the I/O operation. We show the number of cycles taken by the `glibc` double precision exponential function as a point of reference.

The runtime overhead of ViRUS can be divided into one-time and periodic operations. One-time operations include process and knob registration, sensor discovery, and rule parsing. Assuming an app with ten knobs and ten replacement rules, and a processor frequency of 1 GHz, ViRUS adds approximately 0.7 ms

to application initialization. Periodic operations happen every time there is a signal from the variability monitor to the ViRUS handler in the application process. This triggers sensor reading and rule matching (function replacement) in the application, and corresponds to approximately 70 μs under the assumptions above.

5.6.3 Variable Quality Standard Math Library

We implemented a ViRUS library for the mathematical operations of the C standard library declared in the `math.h` header file. The GNU version of the C standard library provides IEEE 754-compliant single and double precision implementations for most math functions [Fou14]. These implementations are typically hardware-accelerated e.g., using SSE (Streaming Single Instruction, Multiple Data) extensions in x86 processors, or VFP (Vector Floating Point) instructions for ARM processors.

For our ViRUS `math.h` implementation, we use the standard double and single precision `libc` functions as quality levels 0 and 1 respectively. For subsequent quality levels, we use implementations from the `fastapprox` [Min14] library. This library provides approximate versions of functions commonly used in machine learning, including exponential, logarithm, power, cos, sin, tan, and others.

Table 5.3 shows total code memory usage for significant methods in the ViRUS math library. To measure code size of each function, we compare the size of the `.text` segment of a statically linked application binary where the application calls only one of the multiple versions and prints the result, with an application that prints a constant number. Because the multiple quality methods may share portions of code, the combined memory usage of the multiple versions of a function is typically smaller than the summation of memory usage for each of the individual versions. Likewise, the combined total for all library functions is less than the

Function	Memory Usage (KBytes)					Combined
	Qual: 0[a]	Qual.: 1[b]	Qual: 2[c]	Qual: 3[d]	Combined	Overhead (%)
exp	31.2	16.2	3.4	3.1	34.7	11.2
log	40.9	15.8	2.8	2.6	44.0	7.6
pow	43.6	18.8	3.7	3.2	50.1	14.9
sin	70.7	9.8	2.7	2.6	79.9	13.0
cos	70.7	9.8	3.0	2.6	79.9	13.0
tan	79.3	9.7	3.1	3.1	88.5	11.7
asin	67.1	16.0	-	-	70.2	4.6
acos	67.1	16.2	-	-	70.3	4.8
atan	37.3	3.6	-	-	39.8	6.9
sinh	33.4	17.7	3.5	3.2	38.6	15.5
cosh	33.4	17.7	3.5	3.2	38.5	15.4
tanh	4.4	4.3	3.5	3.2	8.1	86.7
lgamma	103.5	19.8	3.1	2.8	110.8	7.1
combined	227.8	64.8	4.5	4.1	251.8	10.5

[a] libc double precision
[b] libc single precision
[c] fastapprox best quality
[d] fastapprox worst quality

Table 5.3: ViRUS math library memory usage

summation of the individual functions.

The rightmost column in Table 5.3 shows the overhead of the combined multiple quality methods compared to the single highest quality (double precision) method. The overhead of providing multiple versions of a function over providing only its double precision version averaged 16% and ranged from 5% to 86%. The combined memory usage of all methods on the table was approximately 252 KB, compared to 228 KB for all the methods in double precision only—an increase of 10%. Note that this does not include the small memory overhead of the ViRUS controller/handler library (discussed in Section 5.6.1). We evaluate the impact of the ViRUS math library to application quality and energy usage in Section 5.6.4.

5.6.4 Application case studies

In this section we show how ViRUS can use polymorphic libraries to adjust application quality and energy consumption to counteract system stress from process and environmental variability. We present case studies using the ViRUS math library and show how benchmark applications are impacted in terms of quality and energy across the multiple alternate function implementations.

The potential energy benefits of ViRUS are limited by the fraction of time and energy that an application spends using the polymorphic library functions provided by the system—standard math library functions in our case studies. We profiled a subset of applications in the Parsec suite of benchmarks [Bie11] and found that for some benchmarks that link with the standard numeric library this fraction is too small to yield significant benefits. The `x264` video enconder, for example, relies heavily on integer comparisons. Real-time ray tracing with `raytrace` spends most time and energy in file I/O operations. `Streamcluster` doesn't spend any significant time in standard library functions, with computations for Euclidean distance taking approximately 94% of time in the program.

Simulated annealing with `canneal` uses standard C++ libraries for string comparison, but no significant math functions. Finally, fluid dynamics animation with `fluidanimate` uses floating point math, but doesn't spend significant time in library functions.

In cases where applications do not use significant runtime support system functions, ViRUS may still be useful in managing application-level adaptation. Developers can register multiple versions of a function as knobs by using and linking with the ViRUS controller applications. This modality of application-driven algorithmic choice has been explored in the literature, e.g. in [BC10, LMM07, ACW09]. In our evaluation, we show examples of benchmark applications where library functions represent a significant fraction of application energy and time cost and library implementation impacts application quality: the `whetstone` benchmark, and the `blackscholes` and `swaptions` applications from the Parsec suite.

5.6.4.1 whetstone

Whetstone [CW76] is a standard synthetic benchmark that measures floating-point arithmetic performance. The benchmark combines different operations such as floating point arithmetic, branches, memory access, and procedure calls into a single score. While library math functions comprise only a small fraction of the benchmark, their implementation has dramatic impact in application performance. Library functions used by whetstone include sin, cos, arctan, exponent, and square root.

We modified the Netlib implementation [Rep14] of Whestone to use the ViRUS math library, and profiled energy consumption when each of the different quality levels was used. Energy numbers were obtained by running a nominal instance of VarEMU under nominal operating conditions (voltage, frequency, and tempera-

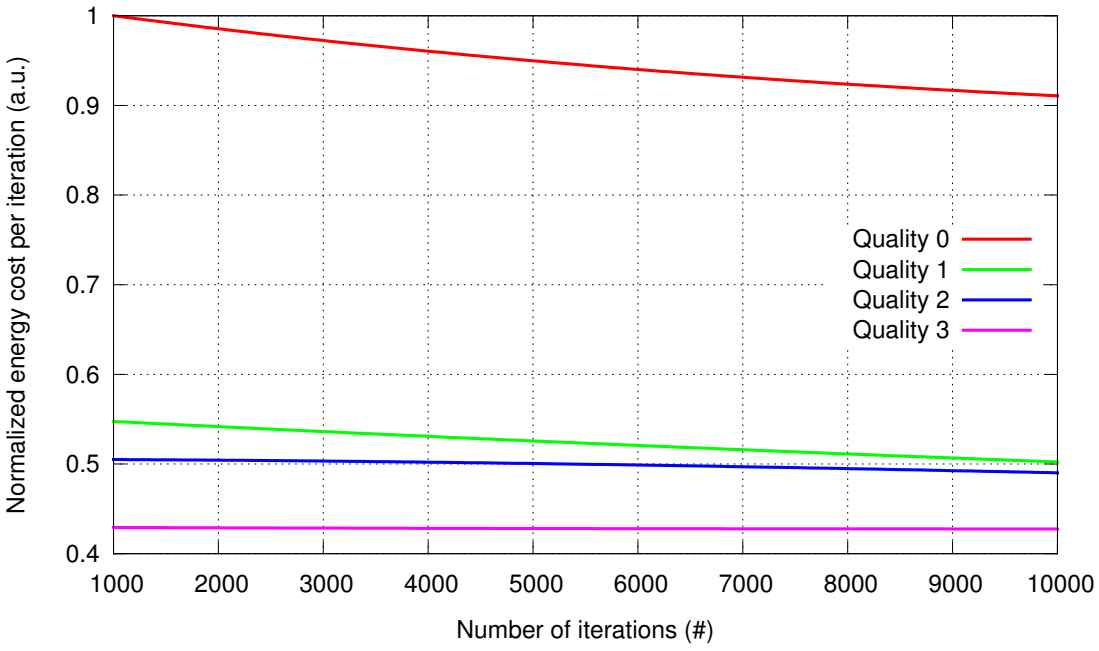


Figure 5.6: Normalized energy cost of Whetstone iterations with different quality versions.

ture). For each run, a global destructor in the application is called upon termination and prints out total accumulated energy. ViRUS configuration files provide rules that lead to immediate switching to a desired quality level upon application initialization. No variation alarms are generated for the run. Energy results are normalized to highest energy cost per iteration (highest quality, smallest number of iterations).

Figure 5.6 shows normalized energy consumption for `whetstone` under different quality levels and across a variable number of iterations for each run. The greatest benefit in energy consumption results from switching from the double precision (highest quality) versions to the single precision versions, which leads to an average 45% reduction in energy consumption. There is a further relative benefit of 8% when going from the single precision version to the first approximate version, and 15% when going from the first approximate version to the lowest quality approximate version. Going from the highest to the lowest quality version

results in a 57% reduction in energy consumption. Because Whetstone does not produce meaningful outputs we do not analyze the impact of the different versions to application quality.

5.6.4.2 blackscholes

The `blackscholes` application is part of the Parsec suite of benchmarks. It computes the Black–Scholes partial differential equation for stock option price estimation [BS73]. Black–Scholes predicts the price of an option on a stock given a certain volatility and interest rate. Because there is no closed-form solution to the partial differential equation, Black–Scholes is computed numerically.

We modified the `blackscholes` application to use the ViRUS math library and profiled its energy consumption as described for `whestone` in Section 5.6.4.1. The implementation of `blackscholes` relies mainly on two numeric library functions, `exp` and `log`. In approximately 500 lines of code, there are two calls to `exp` and one call to `log`, and none of the calls are inside loops for each option price calculation. While `blackscholes` provides runtime options for a varying number of iterations, we found no difference in the relative energy cost of the multiple quality versions as the number of iterations changed.

Figure 5.7 shows normalized energy consumption for `blackscholes` under different quality levels. Going from quality level 0 (double precision) to 1 (single precision) led to a 25% reduction in energy consumption. Going from level 1 to 2 (first approximate version) resulted in a further 18% reduction in energy. From the first to the second approximate version (quality level 2 to 3), there is a 20% energy benefit. Across the board, from the highest to lowest quality version, there is 52% energy consumption band.

Since the Black–Scholes equation yields a meaningful result (the price of an option), we can also analyze the impact of multiple versions to application qual-

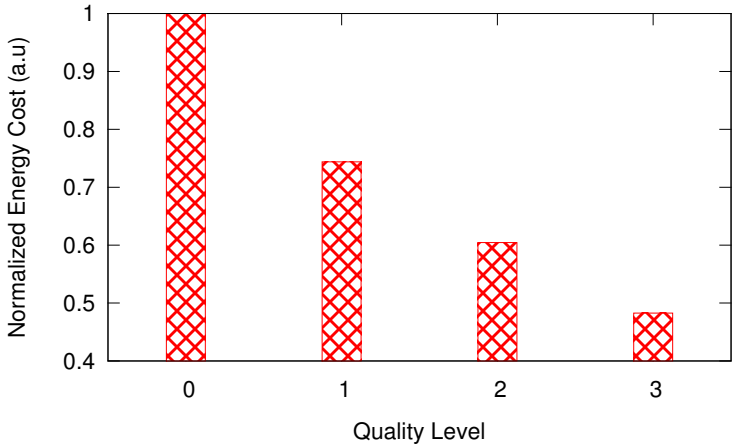


Figure 5.7: Normalized energy cost of `blackscholes` with different quality versions

ity. We ran the benchmark for 64k inputs for each of the quality levels. Figure 5.8 compares the output of the highest quality and approximate results of `blackscholes`. In each of the plots, the x-axis represents the double precision output, and the y-axis is the output when one of the lower quality versions is used.

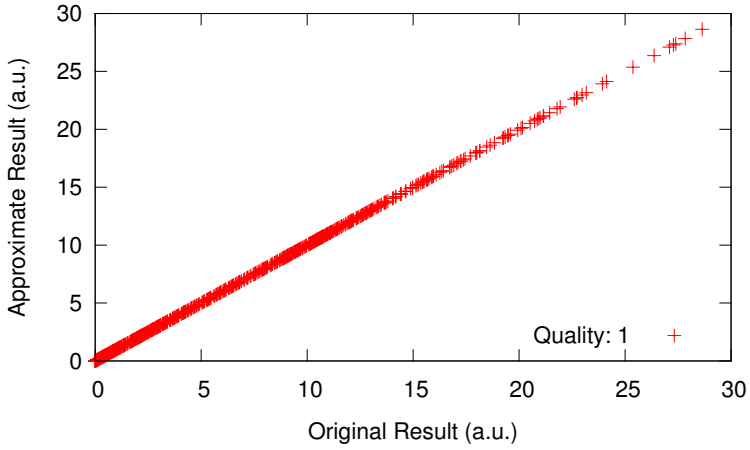
We use two metrics to analyze output quality: normalized root mean square error (NRMSE) and mean absolute percentage error (MAPE). NRMSE is defined as:

$$\text{NRMSE} = 100\% \frac{\sqrt{\frac{\sum_{n=1}^{N}(P_n - A_n)^2}{N}}}{P_{max} - P_{min}} \tag{5.1}$$

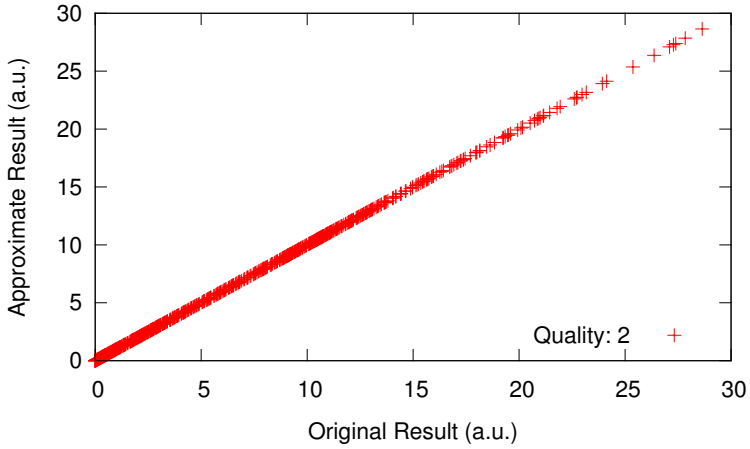
where N is the number of inputs, P_n and A_n are the double precision and approximate outputs respectively for input n, and P_{max} and P_{min} are the maximum and minimum precise outputs across all inputs. MAPE is defined as:

$$\text{MAPE} = \frac{100\%}{N} \sum_{n=1}^{N} \left| \frac{P_n - A_n}{P_n} \right| \tag{5.2}$$

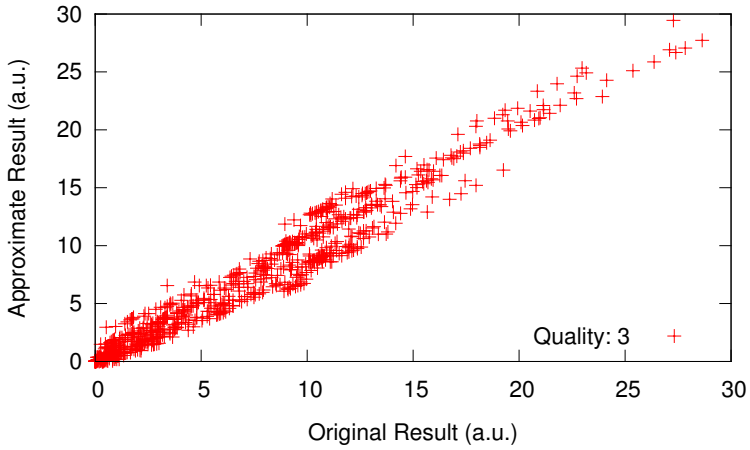
where N is the number of inputs, and P_n and A_n are the double precision and



(a) Quality: 1



(b) Quality: 2



(c) Quality: 3

Figure 5.8: Output for `blackscholes` with different quality levels

Quality	NRMSE (%)	MAPE (%)	(1-NRMSE)/Energy	(1-MAPE)/Energy
0	—	—	1	1
1	0.000005	0.00002	1.34	1.34
2	0.003	0.1	1.65	1.65
3	4.3	39.7	1.98	1.25

Table 5.4: NRMSE and MAPE for `blackscholes`

approximate outputs for input n.

Table 5.4 shows NRMSE and MAPE for `blackscholes` for the multiple qualities in the ViRUS math library, along with measures of energy efficiency. The later numbers are obtaining by dividing $(1 - \text{error})/E$, where error is one of the error metrics, and E is the normalized energy cost for the run. Going across approximate levels in `blackscholes` increases errors by 2–4 orders of magnitude for each step. Acceptable results are produced until step 3, where MAPE becomes approximately 40%. Energy efficiency increases with each step for NRMSE, but is maximized in level 2 for MAPE.

5.6.4.3 swaptions

The `swaptions` benchmark [Bie11] performs a Monte-Carlo simulation to price a portfolio of swaptions. We modified `swaptions` to use the ViRUS math library and profiled its energy usage and output result quality. The core source code module of `swaptions` is an implementation of the Heath-Jarrow-Morton framework for swaption valuation [HJM92]. The exponential function is used three times in approximately 400 lines of code for this module, two of which are within loops. The complete implementation of `swaptions` has approximately 1600 lines of code. As with `blackscholes`, we found no difference in the relative energy cost of `swaptions` across the multiple quality levels as the number of iterations and

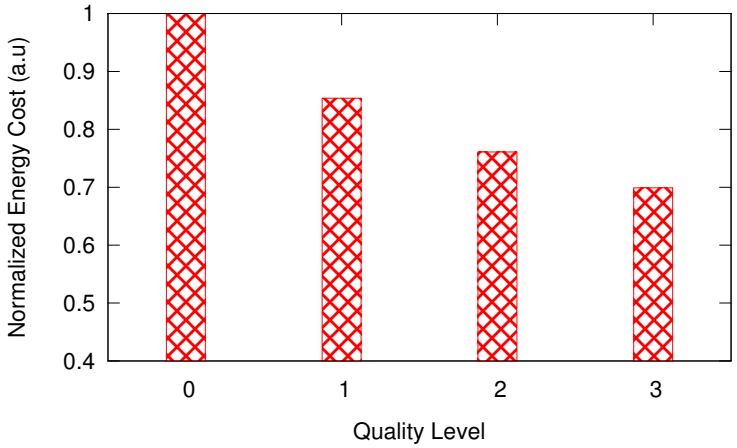


Figure 5.9: Normalized energy cost of `swaptions` with different quality versions

Quality	NRMSE (%)	MAPE (%)	(1-NRMSE)/Energy	(1-MAPE)/Energy
0	—	—	1	1
1	0.0000004	0.000002	1.17	1.17
2	0.002	0.02	1.31	1.31
3	0.57	3.8	1.42	1.37

Table 5.5: NRMSE and MAPE for `swaptions`

inputs of the program varied.

Figure 5.9 shows normalized energy consumption for `swaptions` under different quality levels. Going from quality level 0 (double precision) to 1 (single precision) led to a 15% reduction in energy consumption. Going from level 1 to 2 (first approximate version) resulted in a further 11% reduction in energy. From the first to the second approximate version (quality level 2 to 3), there is a 9% energy benefit. Across the board, from the highest to lowest quality version, there is 30% energy consumption band.

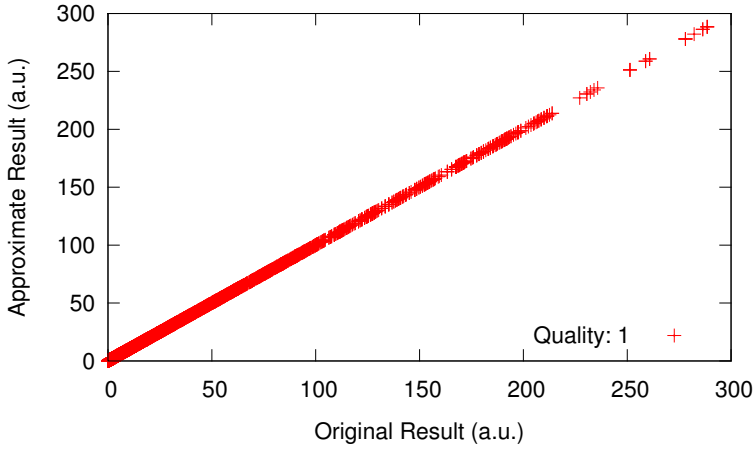
We ran `swaptions` with 4k inputs for each of the quality levels. Table 5.5 shows NRMSE, MAPE, and energy efficiency for `swaptions`. Going across approximate levels in `swaptions` increases errors by 2–5 orders of magnitude

for each step. Acceptable results are produced for all quality levels. In the lowest quality level, MAPE is approximately 40%. Energy efficiency increases with each step for both NRMSE and MAPE. Figure 5.10 compares the output of the highest quality and approximate results.

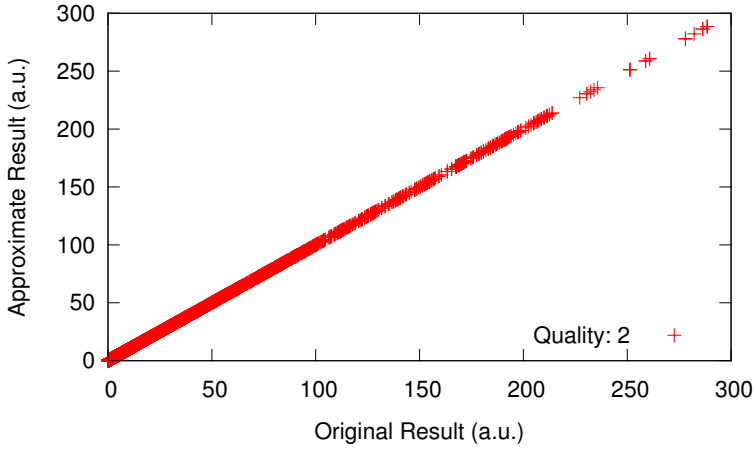
5.7 Discussion

ViRUS provides a band of energy consumption adaptation for applications by supplying multiple quality levels of certain functions and triggering function mutations based on monitoring of energy stress vectors. In this section we discuss some of the limitations of the system.

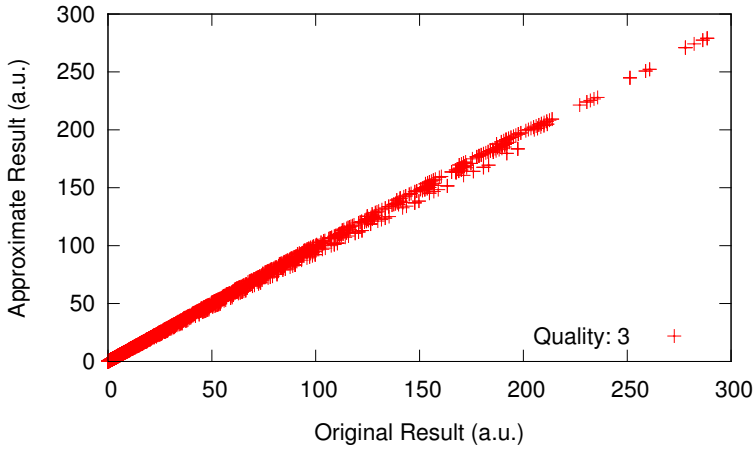
One implicit assumption in the design of ViRUS is that alternate implementations of a function may be ordered in terms of quality and energy cost, i.e. higher cost functions are assumed to produce higher quality results. For certain applications and functions, quality and cost may be input dependent, and hence this ordering may change dynamically. Different quality/cost orders across different applications are easily captured by application-defined function replacement rules in ViRUS. For the same function and sensor range configuration, two applications may for example prefer a different level of a function. If the ordering of quality and cost changes for the same application but different inputs, however, separate configuration files would be required when the application in run with each of the inputs. Application quality feedback mechanisms, e.g. in [BC10] where application developers write a function to evaluate quality of service for a given function and input, burden the programmer and lead to increased runtime overhead due to periodic sampling and re-evaluation of quality/cost for the different app/function/level/input combinations. In future work we instead intend to explore enhanced cost/quality profilers that could automatically assign or suggest mutation rules for applications running under certain environmental and process



(a) Quality: 1



(b) Quality: 2



(c) Quality: 3

Figure 5.10: Output for `swaptions` with different quality levels

variation conditions.

ViRUS functions need not be pure, but must have consistent side effects and state—i.e., each alternative implementation of a function must manipulate any global state consistently and atomically for each function call. For non-pure functions, global variables must be defined in units of code shared by all implementations. This design choice simplifies function mutation both in implementation and in runtime complexity. To allow for inconsistent states, ViRUS would require state migration procedures between every possible pair of alternate functions. Function mutations would also be more costly in runtime—our simpler design choice requires only replacing a function pointer for each migration. While lifting these design restrictions could potentially lead to broader choices of function adaptation, we believe that would lead to an impractical system both in terms of development and runtime overhead.

ViRUS trades off choice in energy consumption for increased memory usage. For the ViRUS math library this memory overhead proved to be very small at approximately 10% added to a monomorphic high precision library. This is further minimized when ViRUS is used in shared libraries, where the overhead may be amortized across many applications. In general, however, the memory overhead of ViRUS can be expected to be potentially as large as the number of choices. Recent trends show the relative energy cost of memory subsystems increasing and leading to more aggressive memory energy management techniques (e.g., [DGN13, BDN12, RBG13]), where certain parts of memory may be powered off in order to decrease overall system energy consumption. In these scenarios, mutations in ViRUS could require dynamic loading and unloading of alternate modules as needed for mutations. We leave an exploration of the costs and benefits of such a scheme for future work.

CHAPTER 6

Conclusions

We started by investigating how much variation exists in contemporary embedded processors. We measured and characterized active and leakage power for a contemporary ARM Cortex M3 processor, and found that across a temperature range of 20–60°C there is 10% variation in active power, and 14x variation in leakage power [WAB10]. Low-end embedded processors like the Cortex M3 are typically a few generations behind state-of-the-art manufacturing processor, so as embedded processors in more advanced technologies become commonplace, the variations will increase significantly.

We introduced different strategies to handle instance and temperature dependent power variation at the system software layer based on task activation control through variability-aware scheduling. We started by formulating the problem of finding the ideal duty cycle rate for a sensor node as an optimization problem subject to power consumption, expected temperature profile, and a lifetime requirement [WBA11]. This was translated into a duty cycle abstraction for TinyOS that allows applications to explicitly specify lifetime and minimum duty cycle requirements for individual tasks, and dynamically adjusts duty cycle rates so that overall quality of service is maximized in the presence of power variability [WAB13]. We showed that variability-aware duty cycling yields a 3–22x improvement in total active time over schedules based on worst-case estimations of power, with an average improvement of 6.4x across a wide variety of deployment scenarios based on collected temperature traces. Conversely, datasheet power specifications fail

to meet required lifetimes by 7–15%, with an average 37 days short of a required lifetime of one year. Finally, we showed that for some applications such as target localization, variability-aware duty cycle scheduling yields a 50% improvement in quality of results over scheduling based on worst-case estimations of power consumption.

We followed up on this framework with VaRTOS, a real-time embedded operating system implementation that optimizes task rewards subject to the same constraints of power consumption, temperature profile, and required lifetime [MWS13]. Tasks in VaRTOS express elasticity by exposing individual *knobs*—shared variables that the operating system can tune to adjust task quality and correspondingly task power, maximizing application utility both on a per-task and system-wide basis. Instead of relying on pre-characterization, VaRTOS dynamically learns instance-specific sleep power, active power, and task-level power expenditure. Our results show that VaRTOS can reduce variability-induced energy expenditure errors from over 70% in many cases to under 2% in most cases and under 5% in the worst-case.

Some classes of embedded sensing applications are not amenable to the task control adaptation described in this work. These include highly synchronized, real-time, or constant data acquisition tasks. Furthermore, our adaptation scheme adds some complexity to the application, in the form of bounds to task activations or knobs, which may in turn lead to further complexities in data storage, inference and communication strategies. Nevertheless, we believe that the benefits of our scheme outweigh the added complexity for a large class of sensing applications. While our duty cycle adaptation scheme indirectly leads to a form of energy-based load balancing across a network of sensors, we do not provide other network-wide adaptation mechanisms such as role selection for nodes, where a node could take different roles (e.g. data collector, router, aggregator) depending on its respective energy rank in the network.

Task activation control is a valuable adaptation strategy for embedded sensing systems, where time spent in sleep mode may account for most of the energy dissipated by the system across its lifetime. For systems with significant variation in active power consumption, a *choice* of software to be executed provides further opportunities for optimization. We introduced ViRUS (Virtual function Replacement Under Stress) as an application runtime support system where the operating system adjusts service quality according to variability-aware policies. In ViRUS, the different code paths provide varying quality-of-service for different energy costs. Mutations from one version to another are triggered by vectors of variability and energy stress. A specific block of code may be activated, for example, when processor temperature reaches a certain threshold. A second block may be activated when remaining battery capacity drops bellow a specified percentage. The different code blocks may be either standard library functions provided by the runtime system or alternative implementations provided by application programmers. Per-application configuration files determine when and under what circumstances code mutations should be triggered. The runtime system monitors the energy stress vector and transparently triggers mutations at appropriate times. We demonstrated ViRUS with a framework for transparent function replacement in shared libraries and a polymorphic version of the standard C math library in Linux. The ViRUS control framework uses less than 3KB of RAM, and the polymorphic math library adds 10% memory overhead to its comparable single choice, high precision version. Application case studies using the polymorphic math library showed how ViRUS can tradeoff upwards of 4% degradation in application quality for a band of upwards of 50% savings in energy consumption.

While developing these software strategies to handle hardware variability, we faced the problem of how to evaluate their effectiveness. Current hardware platforms typically lack variability sensing capabilities. Even if sensing capabilities were available, evaluating variability-aware software techniques across a signifi-

cant number of hardware samples would prove exceedingly costly and time consuming. To cope with this problem, we developed VarEMU, an extension to the QEMU virtual machine monitor that serves as a framework for the evaluation of variability-aware software techniques. VarEMU provides users with the means to emulate variations in power consumption and in fault characteristics and to sense and adapt to these variations in software. Through the use (and dynamic change) of parameters in a power model, users can create virtual machines that feature both static and dynamic variations in power consumption. Faults may be injected before or after, or completely replace the execution of any instruction. Power consumption and susceptibility to faults are also subject to dynamic change according to an aging model. A software stack for VarEMU features precise control over faults and provides virtual energy monitors to the OS and processes. In addition to this work, VarEMU has been used to aid the evaluation of variability-aware software [WEL13] and in graduate-level courses at UCLA.

Code and data supporting this work are available at http://variability.org/ and http://github.com/nesl/.

References

[AAG07] C. Alippi, G. Anastasi, C. Galperti, F. Mancini, and M. Roveri. "Adaptive Sampling for Energy Conservation in Wireless Sensor Networks for Snow Monitoring Applications." In *IEEE Internatonal Conference on Mobile Adhoc and Sensor Systems (MASS)*, pp. 1 –6, oct. 2007.

[ACW09] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. "PetaBricks: a language and compiler for algorithmic choice." *SIGPLAN Not.*, **44**:38–49, June 2009.

[APM05] A. Agarwal, B.C. Paul, S. Mukhopadhyay, and K. Roy. "Process variation in embedded memories: failure analysis and variation aware architecture." *IEEE Journal of Solid-State Circuits*, **40**(9):1804 – 1814, 2005.

[BC10] Woongki Baek and Trishul M. Chilimbi. "Green: a framework for supporting energy-conscious programming using controlled approximation." *SIGPLAN Not.*, **45**:198–209, June 2010.

[BDN12] L. Bathen, N. Dutt, A. Nicolau, and P. Gupta. "Vamv: Variability-aware memory virtualization." In *DATE'12*, 2012.

[BDS11] D. Bull, S. Das, K. Shivashankar, G.S. Dasika, K. Flautner, and D. Blaauw. "A Power-Efficient 32 bit ARM Processor Using Timing-Error Detection and Correction for Transient-Error Tolerance and Adaptation to PVT Variation." *IEEE Journal of Solid-State Circuits*, **46**(1):18–31, 2011.

[Ber06] K. Bernstein et al. "High-performance CMOS variability in the 65-nm regime and beyond." *IBM J. Res. Dev.*, **50**(4):433–449, 2006.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[BKN03] Shekhar Borkar, Tanay Karnik, Siva Narendra, Jim Tschanz, Ali Keshavarzi, and Vivek De. "Parameter variations and impact on circuits and microarchitecture." In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pp. 338–342, New York, NY, USA, 2003. ACM.

[BMR07] Swarup Bhunia, Saibal Mukhopadhyay, and Kaushik Roy. "Process Variations and Process-Tolerant Design." In *20th International Conference on VLSI Design*, pp. 699 –704, jan. 2007.

[BOO06] J. P. Benson, T. O'Donovan, P. O'Sullivan, U. Roedig, C. Sreenan, J. Barton, A. Murphy, and B. O'Flynn. "Car-park management using wireless sensor networks." In *IEEE Conference on Local Computer Networks*, pp. 588–595, 2006.

[BS73] Fischer Black and Myron S Scholes. "The Pricing of Options and Corporate Liabilities." *Journal of Political Economy*, **81**(3):637–54, May-June 1973.

[BSI] BSIM. "http://www-device.eecs.berkeley.edu/∼bsim3/.".

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. "Wattch: a framework for architectural-level power analysis and optimizations." *ACM SIGARCH Computer Architecture News*, **28**(2):83–94, 2000.

[BWV06] S. Bhardwaj, Wenping Wang, R. Vattikonda, Yu Cao, and S. Vrudhula. "Predictive modeling of the NBTI effect for reliable design." In *CICC*, 2006.

[CAC06] L.N. Chakrapani, B.E.S. Akgul, S. Cheemalavagu, P. Korkmaz, K.V. Palem, and B. Seshasayee. "Ultra-Efficient (Embedded) SOC Architectures based on Probabilistic CMOS (PCMOS) Technology." In *Proceedings on the Conference on Design, Automation and Test in Europe (DATE)*, volume 1, pp. 1 –6, march 2006.

[CCF07] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Henrdrik Hamann, Alan Weger, and Pradip Bose. "Thermal-aware task scheduling at the system software level." In *Proceedings of the 2007 international symposium on Low power electronics and design*, ISLPED '07, pp. 213–218, New York, NY, USA, 2007. ACM.

[CDB09] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. "GCS: High-performance gate-level simulation with GPGPUs." In *DATE*, 2009.

[CGK02] Y. Cao, P. Gupta, A.B. Kahng, D. Sylvester, and J. Yang. "Design sensitivities to variability: extrapolations and assessments in nanometer VLSI." In *IEEE International ASIC/SOC Conference*, pp. 411 – 415, sept. 2002.

[CLM12] Hyungmin Cho, L. Leem, and S Mitra. "ERSA: Error Resilient System Architecture for Probabilistic Applications." *IEEE TCAD*, **31**(4):546–558, 2012.

[CLR09] S. Chandra, K. Lahiri, A. Raghunathan, and S. Dey. "Variation-Tolerant Dynamic Power Management at the System-Level." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **17**(9):1220 –1232, 2009.

[CPR04] Seung Hoon Choi, Bipul C. Paul, and Kaushik Roy. "Novel sizing algorithm for yield improvement under process variation in nanometer technology." In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pp. 454–459, New York, NY, USA, 2004. ACM.

[CSG11] Tuck-Boon Chan, John Sartori, Puneet Gupta, and Rakesh Kumar. "On the efficacy of NBTI mitigation techniques." In *DATE*, 2011.

[CW76] H. J. Curnow and Brian A. Wichmann. "A Synthetic Benchmark." *Comput. J.*, **19**(1):43–49, 1976.

[CWC12] Xiaoming Chen, Yu Wang, Yu Cao, Yuchun Ma, and Huazhong Yang. "Variation-Aware Supply Voltage Assignment for Simultaneous Power and Aging Optimization." *IEEE TVLSI*, **20**(11):2143–2147, 2012.

[DBM05] A. Datta, S. Bhunia, S. Mukhopadhyay, N. Banerjee, and K. Roy. "Statistical modeling of pipeline delay and design of pipeline under process variation to enhance yield in sub-100nm technologies." In *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 926 – 931 Vol. 2, march 2005.

[DGA05] Prabal Dutta, Mike Grimmer, Anish Arora, Steven Bibyk, and David Culler. "Design of a wireless sensor network platform for detecting rare, random, and ephemeral events." In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[DGN13] N. Dutt, P. Gupta, A. Nicolau, L.A.D. Bathen, and M. Gottscho. "Variability-aware memory management for nanoscale computing." In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pp. 125–132, Jan 2013.

[DVA10] S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, V. De, and S. Borkar. "Within-Die Variation-Aware Dynamic-Voltage-Frequency Scaling Core Mapping and Thread Hopping for an 80-Core Processor." In *ISSCC*, 2010.

[EKD03] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation." In *Proc. Intl. Symp. Microarchitecture*, pp. 7–18, 2003.

[Fal12] Hossein Falaki. *Automating Personalized Battery Management on Smartphones*. PhD thesis, UCLA, 2012.

[FJ05] Matteo Frigo and Steven G. Johnson. "The Design and Implementation of FFTW3." *Proceedings of the IEEE*, **93**(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[Fou14] Free Software Foundation. "GNU C Library." http://www.gnu.org/software/libc/, 2014.

[Fre13] FreeRTOS Project. "FreeRTOS." http://www.freertos.org, 2013.

[FS99] Jason Flinn and M. Satyanarayanan. "Energy-aware adaptation for mobile applications." *SIGOPS Oper. Syst. Rev.*, **33**:48–63, December 1999.

[GBR07] S. Ghosh, S. Bhunia, and K. Roy. "CRISTA: A New Paradigm for Low-Power, Variation-Tolerant, and Adaptive Circuit Synthesis Using Critical Path Isolation." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **26**(11):1947 –1956, nov. 2007.

[GC07] J. Gregg and T.W. Chen. "Post Silicon Power/Performance Optimization in the Presence of Process Variations Using Individual Well-Adaptive Body Biasing." *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, **15**(3):366 –376, march 2007.

[GKGar] M. Gottscho, A.A. Kagalwalla, and P. Gupta. "Power Variability in Contemporary DRAMs." *IEEE Embedded System Letters*, 2012 (to appear).

[GKM05] Puneet Gupta, Andrew Kahng, and Swamy Muddu. "Quantifying Error in Dynamic Power Estimation of CMOS Circuits." *Analog Integrated Circuits and Signal Processing*, **42**:253–264, 2005. 10.1007/s10470-005-6759-4.

[GM07] Siddharth Garg and Diana Marculescu. "On the impact of manufacturing process variations on the lifetime of sensor networks." In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pp. 203–208, New York, NY, USA, 2007. ACM.

[Goo] Google. "Android Developers Reference." http://developer.android.com/reference/.

[HJM92] David Heath, Robert Jarrow, and Andrew Morton. "Bond Pricing and the Term Structure of Interest Rates: A New Methodology for Contingent Claims Valuation." *Econometrica*, **60**(1):pp. 77–105, 1992.

[HQF10] H. Huang, G. Quan, and J. Fan. "Leakage temperature dependency modeling in system level analysis." In *International Symposium on Quality Electronic Design (ISQED)*, 2010.

[HSC11] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. "Dynamic knobs for responsive power-aware computing." *SIGARCH Comput. Archit. News*, **39**:199–212, March 2011.

[HVY06] T. He, P. Vicaire, T. Yan, L. Luo, L. Gu, G. Zhou, R. Stoleru, Q. Cao, J.A. Stankovic, and T. Abdelzaher. "Achieving Real-time Target Tracking Using Wireless Sensor Networks." *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[ITR] ITRS. "The International Technology Roadmap for Semiconductors." http://public.itrs.net/.

[JKS09] K. Jeong, A.B. Kahng, and K. Samadi. "Impact of Guardband Reduction On Design Outcomes: A Quant. Approach." *IEEE Trans. on Semiconductor Manufacturing*, **22**(4):552–565, 2009.

[KAB03] N. S. Kim, T. Austin, D Baauw, T Mudge, K Flautner, J S Hu, M J Irwin, M Kandemir, and V Narayanan. "Leakage current: Moore's law meets static power." *Computer*, **36**(12):68–75, 2003.

[KHZ07] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B. Srivastava. "Power management in energy harvesting sensor networks." *ACM Trans. Embed. Comput. Syst.*, **6**, 9 2007.

[KPR06] Kunhyuk Kang, Bipul C. Paul, and Kaushik Roy. "Statistical timing analysis using levelized covariance propagation considering systematic and random variations of process parameters." *ACM Trans Des. Autom. Electron. Syst.*, **11**:848–879, October 2006.

[KS07] Vishal Khandelwal and Ankur Srivastava. "Variability-driven formulation for simultaneous gate sizing and post-silicon tunability allocation." In *Proceedings of the 2007 international symposium on Physical design*, ISPD '07, pp. 11–18, New York, NY, USA, 2007. ACM.

[KVR11] Vivek J Kozhikkottu, Rangharajan Venkatesan, Anand Raghunathan, and Sujit Dey. "VESPA: Variability emulation for System-on-Chip performance analysis." In *DATE*, 2011.

[LCB10] L. Leem, Hyungmin Cho, J. Bau, Q.A. Jacobson, and S. Mitra. "ERSA: Error Resilient System Architecture for probabilistic applications." In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 1560 –1565, march 2010.

[LGP07] X. Li, M.J. Garzaran, and D. Padua. "Optimizing Sorting with Machine Learning Algorithms." In *Proc. Parallel and Distributed Processing Symposium*, 2007.

[Lie86] Henry Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object-oriented Systems." *SIGPLAN Not.*, **21**(11):214–223, June 1986.

[LMM07] Andreas Lachenmann, Pedro José Marrón, Daniel Minder, and Kurt Rothermel. "Meeting lifetime goals with energy levels." In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pp. 131–144, New York, NY, USA, 2007. ACM.

[LMP05] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. "TinyOS: An operating system for sensor networks." *Ambient Intelligence*, pp. 115–148, 2005.

[LSL94] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. "Imprecise computations." *Proc. of the IEEE*, **82**(1):83–94, 1994.

[MHY06] Dustin McIntire, Kei Ho, Bernie Yip, Amarjeet Singh, Winston Wu, and William J. Kaiser. "The low power energy aware processing (LEAP)embedded networked sensor system." In *Proceedings of the 5th international conference on Information processing in sensor networks*, IPSN '06, pp. 449–457, New York, NY, USA, 2006. ACM.

[Min14] Paul Mineiro. "fastapprox software library." https://code.google.com/p/fastapprox/, 2014.

[MJ06] Ke Meng and Russ Joseph. "Process variation aware cache leakage management." In *Proceedings of the 2006 international symposium on Low power electronics and design*, ISLPED '06, pp. 262–267, 2006.

[MMF07] Mateusz Malinowski, Matthew Moskwa, Mark Feldmeier, Mathew Laibowitz, and Joseph A. Paradiso. "CargoNet: a low-cost micropower sensor node exploiting quasi-passive wakeup for adaptive asychronous monitoring of exceptional events." In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pp. 145–159, New York, NY, USA, 2007. ACM.

[MTY06] T. Matsuda, T. Takeuchi, H. Yoshino, M. Ichien, S. Mikami, H. Kawaguchi, C. Ohta, and M. Yoshimoto. "A Power-Variation Model for Sensor Node and the Impact against Life Time of Wireless Sensor Networks." In *Proceedings of the First International Conference on Communications and Electronics. (ICCE)*, pp. 106 –111, oct. 2006.

[MWS13] Paul Martin, Lucas Wanner, and Mani Srivastava. "Runtime Optimization of System Utility with Variable Hardware." *ACM Transactions on Embedded Computing Systems*, 2013. Under review.

[NLo13] NLopt Project. "NLopt." http://ab-initio.mit.edu/wiki/index.php/NLopt, 2013.

[NS05] S. Neiroukh and Xiaoyu Song. "Improving the process-variation tolerance of digital circuits using gate sizing and statistical techniques." In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE).*, march 2005.

[PGS11] A. Pant, P. Gupta, and M. v.-d. Schaar. "AppAdapt: Opportunistic Application Adaptation to Compensate Hardware Variation." *IEEE Transactions on Very Large Scale Integration Systems*, 2011.

[PSB05] Ruchir Puri, Leon Stok, and Subhrajit Bhattacharya. "Keeping hot chips cool." In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pp. 285–288, New York, NY, USA, 2005. ACM.

[QEM13] QEMU. "QEMU Open Source Processor Emulator." http://qemu.org, 2013.

[RBG13] Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. "Aging-aware Compiler-directed VLIW Assignment for GPGPU Architectures." In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pp. 16:1–16:6, New York, NY, USA, 2013. ACM.

[RCN96] Jan M Rabaey, Anantha P Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits*, volume 996. Prentice-Hall, 1996.

[Rep14] "Netlib Repository". "Benchmark Programs and Reports." Available at http://www.netlib.org/benchmark/, 2014.

[RGS06] V. Raghunathan, S. Ganeriwal, and M. Srivastava. "Emerging techniques for long lived wireless sensor networks." *IEEE Communications Magazine*, **44**(4):108–114, 2006.

[RGS09] Vijay Janapa Reddi, Meeta S. Gupta, Michael D. Smith, Gu-yeon Wei, David Brooks, and Simone Campanoni. "Software-assisted hardware reliability: abstracting circuit-level challenges to the software stack." In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pp. 788–793, New York, NY, USA, 2009. ACM.

[RSL09] Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. "Apprehending joule thieves with cinder." In *Proceedings of the 1st ACM workshop on Networking, systems, and*

applications for mobile handhelds, MobiHeld '09, pp. 49–54, New York, NY, USA, 2009. ACM.

[SBK06] D. Sylvester, D. Blaauw, and E. Karl. "ElastIC: An Adaptive Self-Healing Architecture for Unpredictable Silicon." *Design Test of Computers, IEEE*, **23**(6):484 –490, 2006.

[SKG07] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. "Eon: a language and runtime system for perpetual systems." In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pp. 161–174, New York, NY, USA, 2007. ACM.

[SN90] Takayasu Sakurai and A Richard Newton. "Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas." *IEEE J. of Solid-State Circuits*, **25**(2):584–594, 1990.

[TKN02] J. Tschanz, J. Kao, S. Narendra, R. Nair, D. Antoniadis, A. Chandrakasan, and Vivek De. "Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage." In *IEEE International Solid-State Circuits Conference (ISSCC)*, volume 1, pp. 422 –478, 2002.

[TST07] A. Tiwari, S. R. Sarangi, and J. Torrellas. "ReCycle: pipeline adaptation to tolerate process variation." In *International Symposium on Computer Architecture*, 2007.

[TT08] Radu Teodorescu and Josep Torrellas. "Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors." In *International Symposium on Computer Architecture*, 2008.

[US14] U.S. Climate Reference Network (USCRN). "Hourly temperature data." http://www.ncdc.noaa.gov/crn/, 2014.

[VE01] Michael J. Voss and Rudolf Eigemann. "High-level adaptive program optimization with ADAPT." *SIGPLAN Not.*, **36**(7):93–102, June 2001.

[Vee84] Harry JM Veendrick. "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits." *IEEE J. of Solid-State Circuits*, **19**(4):468–473, 1984.

[WAB10] Lucas Wanner, Charwak Apte, Rahul Balani, Puneet Gupta, and Mani Srivastava. "A Case for Opportunistic Embedded Sensing in Presence of Hardware Power Variability." In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pp. 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[WAB13] Lucas Wanner, Charwak Apte, Rahul Balani, Puneet Gupta, and Mani Srivastava. "Hardware Variability-Aware Duty Cycling for Embedded Sensors." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **21**(6):1000–1012, 2013.

[WBA11] Lucas Wanner, Rahul Balani, Charwak Apte, Sadaf Sahedi, Puneet Gupta, and Mani Srivastava. "Variability-aware duty cycle scheduling in long running embedded sensing systems." *Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–6, 2011.

[WEL13] Lucas Wanner, Salma Elmalaki, Liangzhen Lai, Puneet Gupta, and Mani Srivastava. "VarEMU: An Emulation Testbed for Variability-aware Software." In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pp. 27:1–27:10, Piscataway, NJ, USA, 2013. IEEE Press.

[WWG08] Geovani Ricardo Wiedenhoft, Lucas Francisco Wanner, Giovani Gracioli, and Antônio Augusto Fröhlich. "Power management in the EPOS system." *SIGOPS Oper. Syst. Rev.*, **42**:71–80, October 2008.

[WYB07] Wenping Wang, Shengqi Yang, Sarvesh Bhardwaj, Rakesh Vattikonda, Sarma Vrudhula, Frank Liu, and Yu Cao. "The impact of NBTI on the performance of combinational and sequential circuits." In *DAC*, 2007.

[Yet93] T. Yetiser. "Polymorphic viruses: Implementation, detection, and protection." Technical report, VDS Advanced Research Group, 1993.

[ZEL02] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. "ECOSystem: managing energy as a first class operating system resource." *SIGOPS Oper. Syst. Rev.*, **36**:123–132, October 2002.

[ZSB10] S. Zahedi, M.B. Srivastava, C. Bisdikian, and L.M. Kaplan. "Quality Tradeoffs in Object Tracking with Duty-Cycled Sensor Networks." In *IEEE Real-Time Systems Symposium (RTSS)*, pp. 160 –169, Dec 2010.

[ZVR09] R. Zheng, J. Velamala, V. Reddy, V. Balakrishnan, E. Mintarno, S. Mitra, S. Krishnan, and Yu Cao. "Circuit aging prediction for low-power operation." In *IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 427–430, 2009.