Runtime Optimization of System Utility with Variable Hardware

PAUL MARTIN, Electrical Engineering Department, University of California, Los Angeles
LUCAS WANNER, Computer Science Department, University of California, Los Angeles
MANI SRIVASTAVA, Electrical Engineering Department, University of California, Los Angeles

Increasing hardware variability in newer integrated circuit fabrication technologies has caused corresponding power variations on a large scale. These variations are particularly exaggerated for idle power consumption, motivating the need to mitigate the effects of variability in systems whose operation is dominated by long idle states with periodic active states. In systems where computation is severely limited by anemic energy reserves and where a long overall system lifetime is desired, maximizing the quality of a given application subject to these constraints is both challenging and an important step toward achieving high-quality deployments. This work describes VaRTOS, an architecture and corresponding set of operating system abstractions that provide explicit treatment of both idle and active power variations for tasks running in real-time operating systems. 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 on a system-wide basis. We provide results regarding online learning of instance-specific sleep power, active power, and task-level power expenditure on simulated hardware with demonstrated effects for several prototypical applications. Our results on networked sensing applications, which are representative of a broader category of applications that VaRTOS targets, 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.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.4.1 [**Operating Systems**]: Process Management—*Threads*; C.3 [**Special-purpose and Application-Based Systems**]—*Real-time and embedded systems*

General Terms: Design, Management, Performance

Additional Key Words and Phrases: Variability, embedded operating systems, power consumption

ACM Reference Format:
Paul Martin, Lucas Wanner, and Mani Srivastava. 2015. Runtime optimization of system utility with variable hardware. ACM Trans. Embedd. Comput. Syst. 14, 2, Article 24 (February 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2656338

1. INTRODUCTION

The emergence of low-power wireless systems in past decades was followed by attempts at optimizing energy efficiency and power consumption to facilitate long lifetime sensing deployments. In recent years, newer integrated circuit fabrication technologies have introduced several additional variables into the energy management game; as

This work is supported in part by the NSF under grants CCF-1029030, CNS-0905580, CNS-0910706, and CNS-1143667.
Authors' addresses: P. Martin, Electrical Engineering Department, 420 Westwood Plaza, 56-125KK EEIV, Los Angeles, CA 90095; email: pdmartin@ucla.edu; L. Wanner, Computer Science Department; email: wanner@ucla.edu; M. Srivastava, Electrical Engineering Department; email: mbs@ucla.edu.

DOI: http://dx.doi.org/10.1145/2656338

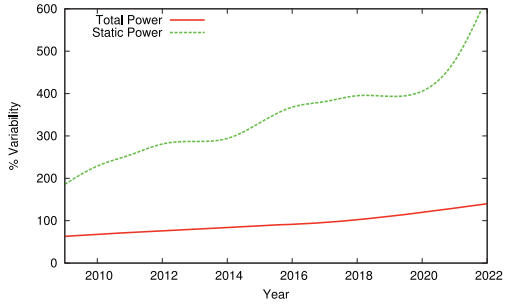

Fig. 1. ITRS predictions for processor power variation for years to come.

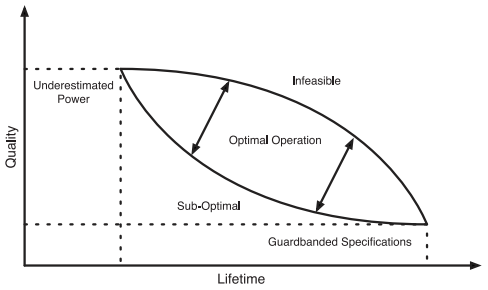

Fig. 2. Potential results of variability in terms of system quality and lifetime.

feature sizes continue to shrink, power variation on a per-instance level has become a nontrivial factor [Borkar et al. 2003; Gupta and Kahng 2003].

Per-instance power variations are particularly exaggerated for idle power consumption, motivating the need to mitigate the effects of variability in systems whose operation is dominated by long idle states. Figure 1 provides more insight into the matter: the International Technology Roadmap for Semiconductors (ITRS) predicts as much as 600% variation in static (idle) power and over 100% in total power by the year 2022 [ITRS 2010]. One domain that stands to benefit from research into combatting hardware variation is that of low-power embedded systems and low-power sensors, where the application is often that of sensing, routing, or processing data.

In systems where computation is severely constrained by anemic energy reserves and where a long overall system lifetime is desired, maximizing the utility of a given application subject to these constraints is both challenging and an important step toward achieving high-quality deployments. Currently, developers assume some power consumption model prior to deployment, and this can have several undesired effects. 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 2, where the optimal solution is one that maximizes quality without decreasing lifetime. Furthermore, the distribution of power in systems composed of multiple heterogeneous tasks is oftentimes fixed in software prior to deployment as well, placing the burden of optimizing energy usage on developers who may remain oblivious to variations in power consumption altogether.

Perhaps the most widely used and most effective strategies for extending the lifetime of energy-constrained systems are those based on controlling the ratio of system active time to total system time, or *duty cycling* a system. Duty-cycled systems take advantage of the disparity between active and idle power consumption, greatly increasing the lifetime of systems where latency and throughput constraints can be relaxed. Because of temperature and instance dependencies in power consumption, however, arriving at an optimal system-wide duty cycle ratio to achieve a lifetime goal given an energy constraint is difficult to do without a priori knowledge of instance-specific power models and temperature statistics for the target deployment location [Wanner et al. 2012]. Furthermore, applications involving more than one task necessitate notions of fairness and utility—specifically, how should active processor time be distributed between each task so as to maximize the utility of the application and still meet the desired lifetime goal?

In this work, we explore the interplay between variable active and idle power consumption, deployment-specific temperature profiles, and multiple heterogeneous tasks. Specifically, we seek an answer to the question posed earlier; in an environment where power and temperature are measurable quantities, we seek an optimal strategy for distributing energy between arbitrary tasks in order to maximize application utility. In answering these questions, we introduce the notion of task *knobs*. These knobs offer both a way for tasks to express elasticity in terms of utility and processing time and a way in which an operating system can fine-tune task energy consumption. Developers provide bounds on the values that each knob can assume and decide in what ways each knob is used, but optimization of these knob values is offloaded to the operating system and is done at runtime after accurate power and computational models have been constructed. These operating system abstractions are implemented in VaRTOS, a variability-aware operating system built as an extension to an open-source embedded operating system. In order to evaluate the abstractions and architectures that make up VaRTOS, we use custom variability extensions to a popular hardware simulation suite.

Our contributions include the following:

—We develop an architecture for modeling and optimizing per-task energy consumption at the operating system level, allowing for tunable quality and, correspondingly, accurate lifetime achievement in the face of variability.
—We provide a tool for evaluating the effects of power variation, environmental temperature, and additional constraints on the quality of user-defined applications composed of multiple tasks prior to system deployment.
—We evaluate VaRTOS, a variability-aware embedded OS that requires a modest 6.8 kB of flash memory and 518 bytes of RAM
—We evaluate the effects of VaRTOS on several prototypical case studies, using a modified version of the QEMU simulation suite [Bellard 2005]. Our results show that VaRTOS can reduce energy consumption error to below 2% in most cases, while strategies that assume worst-case power consumption have greater than 70% error in many cases.

2. RELATED WORK

Hardware-level approaches to address variability have included statistical design approaches [Neiroukh and Song 2005; Datta et al. 2005; Kang et al. 2006], postsilicon compensation and correction [Gregg and Chen 2007; Khandelwal and Srivastava 2007; Tschanz et al. 2002], and variation avoidance [Choi et al. 2004; Bhunia et al. 2007; Ghosh et al. 2007]. Furthermore, variation-aware adjustment of hardware parameters (e.g., voltage and frequency), whether in the context of adaptive circuits (e.g., [Borkar et al. 2003; Ghosh et al. 2007; Agarwal et al. 2005]), adaptive microarchitectures (e.g., [Sylvester et al. 2006; Ernst et al. 2003; Meng and Joseph 2006; Tiwari et al. 2007]), or software-assisted hardware power management (e.g., [Dighe et al. 2010; Chandra et al. 2009; Teodorescu and Torrellas 2008]), has been explored extensively in the literature.

While low-level treatment of hardware variation is a necessary step forward, application- and process-level adaptations have proven to be effective methods for combating variation as well. The range of actions that software can take in response to variability includes altering the computational load by adjusting task activation; using a different set of hardware resources (e.g., using instructions that avoid a faulty module or minimize use of a power-hungry module); changing software parameters (e.g., tuning software-controllable variables such as voltage/frequency); and changing the code that performs a task, either by dynamic recompilation or through algorithmic choice. Examples of variability-aware software include video codec adaptation [Pant

et al. 2012], memory allocation [Bathen et al. 2012], procedure hopping [Rahimi et al. 2012], and error-tolerant applications [Cho et al. 2012]. In embedded sensing, Matsuda et al. [2006] and Garg and Marculescu [2007] provide lifetime analyses for wireless sensor networks when considering variability power models, offering insights into what such systems stand to gain from explicit treatment of hardware variation. Garg and Marculescu estimated that a 37% system lifetime improvement could be achieved through redundancy efforts that totaled a 20% increased deployment cost.

This work attempts to mitigate and exploit variations in power consumption through the management of elasticity in application quality by a variability-aware real-time scheduler. Energy and longevity management in wireless sensor networks and low-power embedded systems in general has long been an active area of research. Most previous work in this field, however, ignores the effects of power variations. Of these variability-agnostic techniques, many have focused on the tradeoff between energy and utility or performance. For example, Baek and Chilimbi [2010] and Ghasemzadeh et al. [2012] represent attempts at making quality energy proportional and tunable. Specifically, Baek and Chilimbi [2010] introduce an architecture that allows developers to specify multiple versions of functions whereby the operating system can sacrifice quality when possible to reduce computational costs. Similarly, Ghasemzadeh et al. [2012] propose tunable feature selection for wearable embedded systems, where less accurate feature computation can be used at the cost of inference quality. In real-time systems, Liu et al. [1994] represent one of many efforts at using approximate computing to save energy where marginal losses in quality can be afforded. In ECOSystem [Zeng et al. 2002] and Cinder [Rumble et al. 2009], energy resources are periodically distributed to tasks that must spend the resources to perform system calls. In these systems, applications adjust their computational load according to energy availability. Our work differs from previous approaches in that applications need not manage energy directly but instead expose their elasticity in the form of a variable knob that is controlled by the operating system scheduler. Power consumption characteristics for each individual task are learned over time, and the system maximizes quality of service across tasks in a variability-aware fashion.

This work is closely related to that of Wanner et al. [2012]. There, the authors describe a method for calculating a system-wide optimal duty cycle ratio given known models for active and idle power as well as probability density functions for deployment temperatures. Here we provide an extension to the work in Wanner et al. [2012], showing methods for online learning of power models and providing notions of utility in multitask applications.

3. POWER VARIABILITY

As fabrication technologies improve and feature sizes decrease, hardware variation plays an increasingly important role in determining the power consumption and therefore lifetime of computer systems. This variation can be attributed to manufacturing (due to scaling of physical feature dimensions faster than optical wavelengths and equipment tolerances [Bernstein et al. 2006; Cao et al. 2002]), environment (e.g., voltage and temperature), aging (e.g., due to negative bias temperature instability [Zheng et al. 2009]), and vendors (multisourcing of parts with identical specifications from different manufacturers).

Power consumption in an embedded processor can be classified as either active power or sleep (idle) power. Active power includes switching and short circuit power and can be modeled as in Rabaey et al. [1996] and Veendrick [1984]:

$$P_a = CV_{dd}^2 f + \eta(V_{dd} - V_{thn} - V_{thp})^3 f, \tag{1}$$

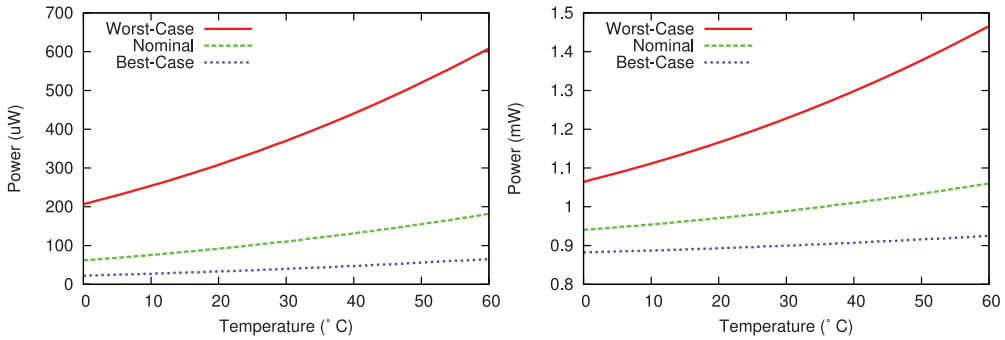

Fig. 3. Sleep (left) and active (right) power for three processor instances in 45nm representing the range of variation for the technology.

where C is the switching capacitance, V_{dd} is the operating voltage, f is the clock frequency, η is a technology- and design-dependent parameter, V_{thn} is the threshold voltage for NMOS, and V_{thp} is the threshold voltage for PMOS. The threshold voltage V_{thp} is subject to wearout due to negative bias temperature instability (NBTI) as described in Chen et al. [2012], Bhardwaj et al. [2006], and Wang et al. [2007]. Sleep power can be modeled as

$$P_s = V_{dd}(I_{sub} + I_g), \tag{2}$$

where I_{sub} is the subthreshold leakage current and I_g is the gate leakage current. Subthreshold leakage current models can be derived from the device model in UC Berkeley Device Group [2013], and simplified to extract its temperature and voltage dependency:

$$I_{sub} = a_1 T^2 \left(\exp\left(\frac{-a_2 V_{thp}}{T} \right) + \exp\left(\frac{-a_2 V_{thn}}{T} \right) \right) \exp\left(\frac{-a_3 V_{dd}}{T} \right), \tag{3}$$

where T is temperature in Kelvin and $\{a_1, a_2, a_3\}$ are empirically fitted parameters that capture part-to-part variations. Gate leakage current is defined in Kim et al. [2003] as:

$$I_g = a_4 V_{dd}^2 \exp(-a_5 / V_{dd}), \tag{4}$$

where a_4 and a_5 are empirically fitted parameters.

While the large baseline in active power consumption relative to idle power consumption amortizes variations to some degree (Wanner et al. [2012] cite a 10% variation in active power while Balaji et al. [2012] cite between 7% and 17% variation), the low baseline in idle power consumption renders it highly susceptible to fabrication-induced variations (Wanner et al. [2012] report a 14 times range in measured idle power across 10 instances of ARM Cortex M3 processors in 130nm technology).

In this work, we use 45nm process technology and libraries as our baseline for evaluation. Power model parameters are fitted to the SPICE simulation results of an inverter chain using the device model given in the technology libraries. The final power values are normalized to the measured data obtained from an M3 test chip using the same technology. Figure 3 shows active and sleep power across temperature for three instances representing the range of power variation for this technology (nominal, worst case, and best case). At room temperature, there is approximately 6 times variation in sleep power consumption between the worst- and best-case instances. This magnitude of variation matches measurements with off-the-shelf embedded class processors fabricated in 130nm shown in Wanner et al. [2012], and hence represents a conservative estimation of the variation that may be found in processors in newer technologies.

Power consumption in energy-constrained embedded systems is often dominated by time spent in an idle state. Consequently, the lifetimes of these systems can be widely variant due to instance-to-instance variation in idle power, resulting in either premature system death or suboptimal system quality and an energy surplus.

4. OPTIMIZING UTILITY WITH VARIABLE HARDWARE

One way to combat increasing power variability is for an application to decrease or increase quality, thereby decreasing or increasing energy consumption. In this section, we explore an architecture for adapting quality when applications have some degree of elasticity—that is, when quality is not a hard constraint. This will be accomplished by introducing task *knobs*—task-specific expressions of quality and power elasticity.

4.1. Task Modeling

We start by introducing an application as a set of N tasks denoted $\tau_i, i \in \{1, \ldots, N\}$, where each task represents a periodic application subprocess. We associate with each task and with the application as a whole a utility u_i (or u_{sys} for the entire application) with the understanding that utility represents some notion of quality that the user is interested in. While some efforts espouse an architecture wherein each u_i is defined by an arbitrary function (e.g., [Baek and Chilimbi 2010]), we advocate a simplified model where the OS constructs u_i based on a few key inputs from the developer. In doing so, we assume that u_i is a monotonically nondecreasing function of the active computational time for τ_i denoted $t_{a,i}$ or, equivalently, the duty cycle ratio specific to τ_i denoted d_i and defined as $d_i = t_{a,i}/(t_{a,i} + t_{s,i})$, where $t_{s,i}$ is the amount of time that τ_i is inactive. These variables along with additional key variables used throughout the text are summarized in Table I.

Task Knobs: In order to tune the active time used per task and thus the task-specific duty cycle ratio d_i, we introduce the notion of task knobs. In practical terms, a task knob is a variable that will govern either (1) the period of a task or (2) the frequency with which a task is activated. We argue that a large portion of tasks found in embedded applications will fall into one of these two classes, and those that require both frequency and period modulation can often be divided into two legal subtasks coupled with interprocess 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, and more.

We define task knobs, denoted $k_i \in \mathbb{Z}^+$, such that increasing k_i will increase $t_{a,i}$, d_i, and, consequently, 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.

Generating Utility Curves: Changing each knob value k_i will cause a corresponding change in duty cycle ratio d_i based on the nature of τ_i. In general, utility functions $u_i = f(d_i)$ can be arbitrary. In practice, however, increasing the duty cycle of a task indefinitely will lead to diminishing returns on overall system utility. Because of this, we use a general model of utility based on the convex portion of a logistic function. The characteristic s-like curve of logistic functions offers a convenient form for modeling

Table I. Summary of Selected Variables

Variable Name	Definition
τ_i	Task $i, i \in \{1, \ldots, N\}$
$t_{a,i}$	Active time for τ_i
$t_{s,i}$	Inactive time for τ_i
d_i	D.C. ratio for τ_i
$\boldsymbol{d}$	The vector $[d_1 \cdots d_N]$
d_{sys}	System-wide D.C.
k_i	Knob value for τ_i
$\boldsymbol{k}$	The vector $[k_1 \cdots k_N]$
$\mathcal{K}_i$	Model of $k_i \rightarrow d_i$
u_i	Utility fcn. for τ_i
p_i	Priority scalar for τ_i
E	Energy budget
L	Desired lifetime

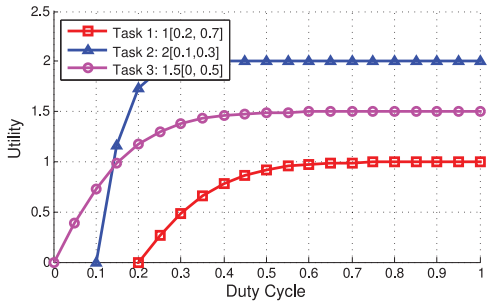

Fig. 4. Example utility curves. Task 1 has priority scalar $p_1 = 1$ with useful range $d_1 = [0.2, 0.7]$, task 2 with $p_2 = 2$ and $d_2 = [0.1, 0.3]$, and finally task 3 with $p_3 = 1.5$ and $d_3 = [0, 0.5]$.

diminishing returns on d_i. Given $k_{i,min}$ and $k_{i,max}$ as well as a mapping from k_i to d_i (to be discussed later), we can construct a utility function $u_i = f(d_i)$ as a modified logistic (Sigmoid) function of the form $f(d_i) = \frac{1}{1+e^{-c_i d_i}}, c_i \geq 0$. The convex portion of the logistic function can be isolated by choosing u_i to be of the particular form

$$u_i(d_i) = \frac{2}{1 + e^{-c_i d_i}} - 1, \quad c_i \geq 0, \quad d_{i,min} \leq d_i \leq d_{i,max}, \tag{5}$$

where $d_{i,min}$ and $d_{i,max}$ are task duty cycles corresponding to $k_{i,min}$ and $k_{i,max}$. 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}. Increasing the percentage of $u_{i,max}$ realized by $k_{i,max}$ has the effect of steepening the utility curve and thus increasing the rate at which returns diminish. For nonadaptable tasks (i.e., $d_{i,min} = d_{i,max}$ or where k_i is unused), u_i is no longer a function of d_i and is therefore set to a constant value. When $d_{i,min} \neq d_{i,max}$, the constant c_i can be calculated from Equation (5) by enforcing $u_i(d_{i,max}) - u_i(d_{i,min})$ to be $\epsilon = 0.99$ as shown here:

$$c_i = \frac{-\log\left(\frac{2}{\epsilon+1} - 1\right)}{(d_{i,max} - d_{i,min})}, \quad \epsilon = 0.99. \tag{6}$$

Finally, each utility curve can be arbitrarily increased or decreased by a priority scalar $p_i \in \mathbb{R}^+$ for tasks with intrinsically higher or lower utility than others. This offers a level of customizability in addition to specifying $k_{i,min}$ and $k_{i,max}$, allowing the developer to give preference to one task over another. Figure 4 shows three example utility curves corresponding to three tasks with various priorities and duty cycle ranges (resulting from various $k_{i,min}$ and $k_{i,max}$).

Learning $\mathcal{K}_i$, *the* $k_i \rightarrow d_i$ *Relation*: Because the developer is free to use the knob k_i for each task as desired, the function mapping k_i to active time $t_{a,i}$ and thus d_i is not known a priori. Instead, the transformation $\mathcal{K}_i$ that maps k_i to d_i is assumed linear and is learned through regression at runtime. Should the developer misuse k_i in a way that is nonlinear or that results in nonincreasing values of $t_{a,i}$, the linear model will introduce errors that will affect the optimization process. Dividing active time accumulated per task by a fixed supervisory time interval t_{super} yields task-specific duty cycle ratios, d_i.

4.2. Maximizing Application Utility

Given the set of tasks $\{\tau_1, \ldots, \tau_N\}$, our ultimate goal is to optimize $u_{sys} = \sum_{i=1}^{N} u_i$, the overall system utility. That is, we seek a solution to the convex optimization problem

$$u^*_{sys} = \max_{\boldsymbol{k}} \sum_{i=1}^{N} \frac{2}{1 + e^{-c_i \mathcal{K}_i[k_i]}} - 1, \tag{7}$$

$$\text{subject to: } \sum_{T=T_{min}}^{T_{max}} f_T \mathcal{L}[\boldsymbol{k}, T] \leq \frac{E}{L} = \bar{P},$$

where T_{min} and T_{max} are the minimum and maximum temperatures for a given location, respectively; $\boldsymbol{k} = [k_1 \cdots k_N]$ is the vector of task knobs; and $\mathcal{L}$ is a mapping from $\boldsymbol{k}$ to power consumption. This mapping is assumed linear for a given temperature T. The parameters E and L are the energy budget and desired lifetime as specified by the user and resulting in an average power goal, $\bar{P}$. In general, $\mathcal{L}$ is a function of temperature, and thus summing over the range of temperatures $[T_{min}\ T_{max}]$ and scaling by the probability mass function of each temperature f_T (obtained by discretizing temperature T into bins a priori as described in Section 5.4) give the predicted power consumption corresponding to the knob vector $\boldsymbol{k}$. When using external peripherals (such as radios, analog-to-digital converters, and flash storage), $\mathcal{L}$ incorporates both external power and internal power (i.e., power consumed by the processor). In the remainder of this article, we will focus on optimizing utility when $\mathcal{L}$ includes the power consumption of the processor alone. We leave inclusion of peripheral power models as a natural and straightforward extension to the proposed architecture.

We shift our focus now from that of optimizing utility with a power constraint to that of optimizing utility with a duty cycle constraint. In other words, power consumption of the system as a whole can take on values in the range $[P_s(T)P_a(T)]$ for a given temperature T dictated by the overall system duty cycle ratio, $d_{sys} = \sum_{i=1}^{N} d_i$: $P = (1 - d_{sys})P_s + d_{sys}P_a$. Again, both P_s and P_a are functions of temperature so that, replacing $\mathcal{L}$ with the processor power models, the optimization problem becomes

$$u^*_{sys} = \max_{\boldsymbol{d}} \sum_{i=1}^{N} \frac{2}{1 + e^{-c_i d_i}} - 1, \tag{8}$$

$$\text{subject to: } \sum_{T=T_{min}}^{T_{max}} f_T \left[\sum_{i=1}^{N} d_i P_{a,i}(T) + \left(1 - \sum_{i=1}^{N} d_i\right) P_s(T) \right] \leq \frac{E}{L} = \bar{P}.$$

Here we have replaced the knob vector $\boldsymbol{k}$ with the duty cycle vector $\boldsymbol{d}$ similarly defined. The system-wide duty cycle can be arrived at if we know a priori the future environmental temperatures, the function mapping temperature to sleep power $P_s(T)$, and the function mapping temperature to active power for a given task $P_{a,i}(T)$. For most embedded class processors, active power consumption does not vary significantly across instructions so that the task-specific $P_{a,i}(T)$ can be replaced by a system-wide $P_a(T)$. Following from Equation (8), the optimal (maximum) system-wide duty cycle $d^*_{sys} \in [0, 1]$ can be formulated as a maximization over a variable d:

$$d^*_{sys} = \max\ d \tag{9}$$

$$\text{subject to: } \sum_{T} f_T \left[dP_a(T) + (1 - d)P_s(T) \right] \leq \frac{E}{L} = \bar{P}.$$

ALGORITHM 1: Greedy Utility Optimization

Input: System duty cycle, d^*_{sys}, linear functions $\{\mathcal{K}_1, \ldots, \mathcal{K}_N\}$, and utility curves $\{u_1, \ldots, u_N\}$
Output: The optimal task duty cycles $\boldsymbol{d}^* = \{d^*_1, \ldots, d^*_N\}$
$\boldsymbol{\tau} \leftarrow \text{sort}(\{\tau_1, \ldots, \tau_N\})$ by decreasing p_i
$d_{remaining} \leftarrow d^*_{sys}$.
// Assign minimum knob values for tasks that can be scheduled:
$\boldsymbol{\tau}_{scheduled} \leftarrow \{\tilde{}\}$ // empty set
for $i \in \boldsymbol{\tau}$ **do**
  if $\mathcal{K}_i[k_{i,min}] < d_{remaining}$ **then**
    $d_i = \mathcal{K}_i[k_{i,min}]$
    $d_{remaining} \leftarrow d_{remaining} - d_i$
    append τ_i to $\boldsymbol{\tau}_{scheduled}$
  else
    stop
  end
end
// Allocate remaining duty cycle fairly:
while $d_{remaining} > 0$ **do**
  // Find highest marginal utility *max_mu* and the set $\boldsymbol{\tau}_{max}$ of tasks yielding *max_mu*:
  $[max\_mu, \boldsymbol{\tau}_{max}] \leftarrow$ Find Maximum Marginal Utility($\boldsymbol{\tau}_{scheduled}$)
  $d_{requested} \leftarrow \min\{\delta \cdot |\boldsymbol{\tau}_{max}|, \tilde{} d_{remaining}\}$
  for $i \in \boldsymbol{\tau}_{max}$ **do**
    $d_i \leftarrow d_i + \frac{1}{|\boldsymbol{\tau}_{max}|} d_{requested}$
  end
end
$\boldsymbol{d}^* \leftarrow \{d_1, \ldots, d_N\}$

Under practical conditions as outlined in Wanner et al. [2012], a close approximation for the optimal solution to Equation (9) can be obtained algebraically using Equation (10), where we have introduced the temperature-averaged power quantities $\bar{P}_s$ and $\bar{P}_a$:

$$d^*_{sys} = \frac{E - L\bar{P}_s}{L(\bar{P}_a - \bar{P}_s)}. \tag{10}$$

Given d^*_{sys}, we now seek an efficient solution to Equation (8). Because we have chosen u_i to be a logistic function, we can use a greedy approach when optimizing utility. The optimization routine will be a two-step process: (1) attempt to assign the minimum duty cycle $d_{i,min} = \mathcal{K}_i[k_{i,min}]$ 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. This process is outlined in Algorithm 1. Duty cycle is incrementally added by a small fraction δ (chosen sufficiently small to ensure accuracy) to those tasks with the largest marginal utility defined as

$$max\_mu = \max\{mu_1, \ldots, mu_N\}, mu_i = \frac{u_i[d_i + \delta] - u_i[d_i]}{\delta} \tag{11}$$

until d^*_{sys} as calculated from Equation (10) is exhausted. Figure 5 shows an example of the utility maximization algorithm for two example tasks with the total system duty cycle on the x-axis and utility on the y-axis. At point (a), $k_{1,min}$ is met and task 1 begins to receive active time. At point (b), task 1 begins to plateau as mu_1 diminishes. At point (c), both $k_{1,min}$ and $k_{2,min}$ can be met; starting with task 1 with the highest *mu*, utility is

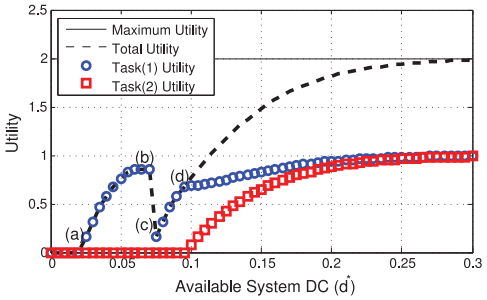

Fig. 5. Maximizing utility for two tasks and different values of system duty cycle, d^*_{sys}.

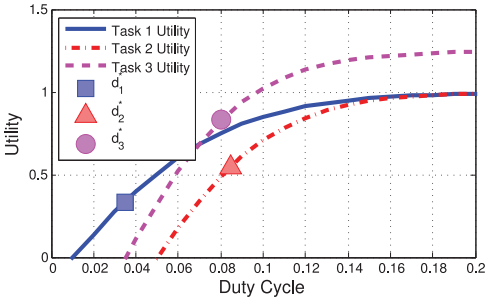

Fig. 6. Example optimal duty cycle points for three example tasks with different values for $\{k_{min}, k_{max}, p_i\}$.

increased until point (d), where task 2 and 1 go back and forth bidding for active time. The dashed curve illustrates the total system utility, $u_{sys} = u_1 + u_2$.

Each point in Figure 5 represents a different environmental setup—that is, a different d^*_{sys} (x-axis) resulting from, perhaps, different values for E, L, and $\boldsymbol{f}_T$. At each d^*_{sys}, all tasks are assigned a specific duty cycle ratio d_i. For example, Figure 6 shows the resulting duty cycles $\{d_1, d_2, d_3\}$ for three tasks when $d^*_{sys} = 0.2$. The vector $\boldsymbol{d}$ that maximizes Equation (8) is denoted $\boldsymbol{d}^* = \{d^*_1, \ldots, d^*_N\}$.

With a method in hand to calculate an optimal d^*_{sys} offline, we seek a method for both calculating d^*_{sys} and achieving $d_i \in \{d_1, \ldots, d_N\}$ in an efficient, online manner. Our implementation of this architecture is called VaRTOS and is the subject of the following section.

5. VARTOS, THE VARIABILITY-AWARE REAL-TIME OPERATING SYSTEM

In this section, we outline the implementation of the architecture and algorithms presented in Section 4 as a series of extensions to an existing real-time operating system (RTOS). The results shown use a modified version of the FreeRTOS operating system [FreeRTOS Project 2013], though the architecture is easily applied to other embedded operating systems as well. The design of VaRTOS must accomplish several key aspects of Section 4 while remaining lightweight and energy efficient. In particular, VaRTOS includes the following functionality: (1) a method for online modeling of $P_s(T)$ and $P_a(T)$, (2) a method for online modeling of $\mathcal{K}_i$, (3) OS-level control over task knobs $\{k_1, \ldots, k_N\}$, and (4) a tool for evaluating the effects of user inputs $\{k_{i,min}, k_{i,max}, p_i, E, L\}$ as well as deployment location (temperature profile). We will describe each of these subsystems in detail next, with various prototypical case studies discussed in Section 8.

5.1. Online Modeling of Sleep and Active Power

As discussed in Section 3, both sleep power and active power are nonlinear functions of temperature. The vast majority of this nonlinearity comes from leakage and subthreshold currents that dominate in P_s. In general, modeling these nonlinear curves 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 300kB of program space in order to do even rudimentary optimization routines [NLopt Project 2013] and prohibiting its use in many low-power platforms. Fortunately, the models in Section 3 describing P_s result in a function that is very closely exponential. Knowledge

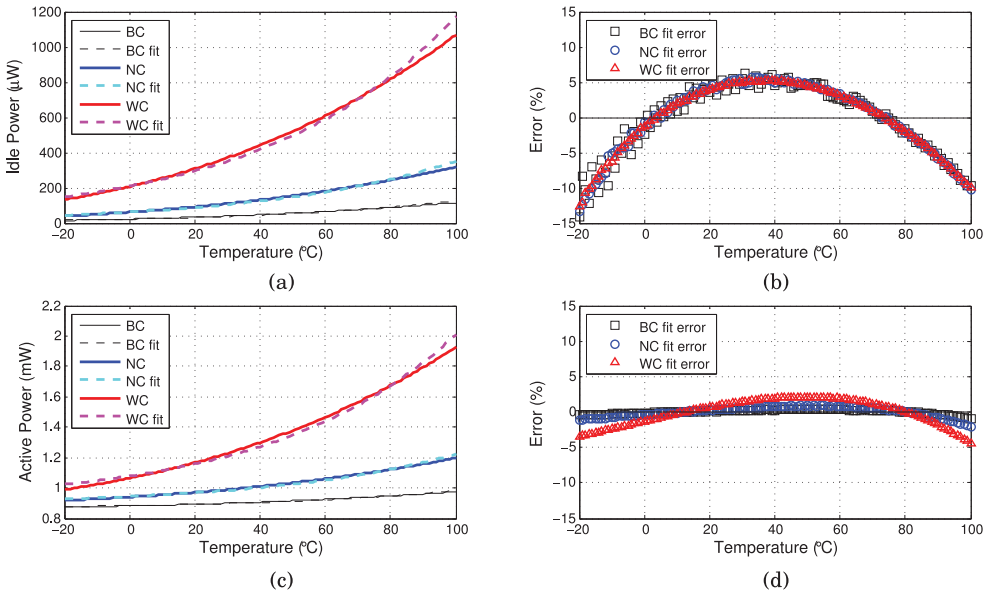

Fig. 7. Modeling sleep and active power through linearization.

of the shape of this function 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 Section 3 and the linear approximation methods described earlier is shown in Figure 7 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 temperate range of [0°C, 80°C], the worst-case error is around 5%. For most temperature 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 d^*_{sys} 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 $\sim \mathcal{N}(0, 5\mu\text{W})$, 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 8 for 190 different locations within

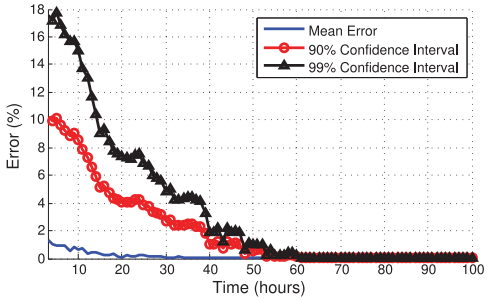

Fig. 8. Error convergence for sleep power modeling. Shown are the mean errors, 90% confidence, and 99% confidence intervals for errors versus the steady-state models.

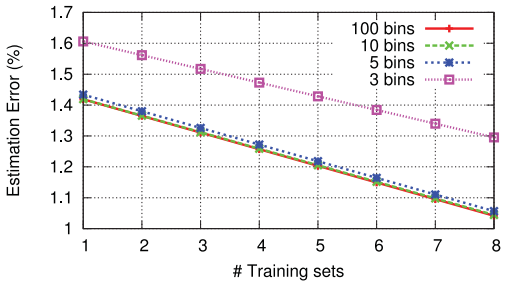

Fig. 9. Error in average power estimation for temperature models constructed from varying numbers of training years (x-axis) and with a set number of histogram bins.

the United States with between 1 and 9 years of hourly data in all locations and for processor instances with best-case, nominal-case, and worst-case power consumption. For the results that follow, both sleep and power models will be fit after 40 points (40 hours) of data have been collected.

5.2. Online Modeling of Task Computation

Active time per task $t_{a,i}$ can be measured in a number of ways (e.g., using hardware timer snapshots at the context swap level). Given a method for measuring $t_{a,i}$, $\mathcal{K}_i$ is arrived at by systematic perturbation of k_i within the range $[k_{i,min}, k_{i,max}]$. Specifically, k_i is repeatedly increased by $\Delta = \frac{k_{i,max} - k_{i,min}}{n}$, where n is the number of points in the regression, kept sufficiently low ($n = 4$ in our case) to minimize memory footprint. Between each perturbation in k_i, the task is allowed to run for a time period sufficiently long enough to accurately model tasks with infrequent activity or control-flow-dependent variations in execution time. In the applications presented here, this supervisory period is set at $t_{super} = 1$ hour, meaning the mappings $\mathcal{K}_i$ are calculated after 4 hours. Task duty cycles are calculated as $d_i = (\sum t_{a,i})/t_{super}$. Note that $\mathcal{K}_i$ is a linear transformation from k_i to d_i and thus k_i should translate linearly into active time for that task. In Section 8, we explore the effects of violating this assumption.

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. For example, on entering a subroutine, the `taskEnterISR(taskHandle)` command is invoked with a matching `taskExitISR` upon finishing the subroutine. Again, as mentioned in Section 4.2, in some cases additional peripheral power will be expended during these subroutines. The metric $t_{a,i}$ reflects only processor power expenditure, and thus peripheral power usage must be modeled separately by modifying $\mathcal{L}_i$.

5.3. Controlling Task Active Time

In the same way that k_i is perturbed to model $\mathcal{K}_i$ in the previous section, k_i is also commanded by the operating system to achieve d_i^* as calculated by Algorithm 1. Knob control is passed from user to operating system at task creation, making the full task creation call using the modified FreeRTOS kernel:

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

By its nature, TaskKnob serves as a discrete representation of k_i and therefore introduces quantization errors into the optimization routine. In particular, the smaller the difference between $k_{i,min}$ and $k_{i,max}$, the coarser the granularity of TaskKnob becomes and therefore the poorer the achievable resolution of d_i becomes. As an extreme example, if $k_{i,min} = 1$ and $k_{i,max} = 2$, then TaskKnob can only take on one of two values and thus one of two d_i values, perhaps far away from d_i^*. Similarly, even if TaskKnob is constructed in such a way that it has fine granularity, d_i^* might not be within the range $[\mathcal{K}_i[k_{i,min}], \mathcal{K}_i[k_{i,max}]]$. When d_i^* is less than $\mathcal{K}_i[k_{i,min}]$, task τ_i consumes more energy than it is allotted and the system is likely to die prematurely. If d_i^* is greater than $\mathcal{K}_i[k_{i,max}]$, however, it may simply mean that even though additional energy can be allotted to task τ_i, no additional utility would be gained and so achieving a lifetime greater than L is acceptable.

5.4. Temperature Models

Equations (9) and (10) require that we know the temperature distribution $\boldsymbol{f}_T$ in order to calculate d_{sys}^* and the individual task ratios $\{d_1^*, \ldots, d_N^*\}$. We performed several simulations with results indicating that learning $\boldsymbol{f}_T$ online is infeasible, as it takes the entirety of a year to develop an accurate histogram of the temperature values seen at a given location. Fortunately, similar simulations show that a very coarse representation of the temperature profile suffices for accurate calculations of d_{sys}^*, and furthermore temperature profiles change very little from year to year for a given location. Figure 9 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 $\boldsymbol{f}_T$ for a given location. This figure makes two noteworthy points: first, the decreasing estimation error indicates that temperature profiles change very little from year to year, and because of this using multiple years to build $\boldsymbol{f}_T$ only serves to decrease the prediction error in years to come; second, while a three-bin histogram is inadequate to fully represent the temperature profile for a given location, there is very little benefit in representing $\boldsymbol{f}_T$ with more bins than five 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 $\boldsymbol{f}_T$.

5.5. User Programming Model

Much of the effort in creating VaRTOS is in making the process transparent to the developer and easing the burden of accounting for variable task power consumption. In addition to the challenges that come with embedded programming in general, developers need only provide the following information: (1) energy budget E measured in joules; (2) lifetime goal L measured in hours; (3) deployment location if it belongs in the VaRTOS database or corresponding coarse $\boldsymbol{f}_T$ if not; and (4) $k_{i,min}$, $k_{i,max}$, and priority scaling factors p_i.

Information required from the developer is therefore very minimal, though in many circumstances it is not readily apparent how different user inputs—particularly for knob values and priorities—will affect the operation of the system. In order to provide more intuition regarding the various parameters the developer is tasked with supplying, we have developed a simple tool in MATLAB as shown in Figure 10. This tool allows the developer to specify an energy budget and lifetime goal to guide the optimization process. Developers further specify clock frequency, instance type, a certain geographical location, and the various tasks to be scheduled. Perhaps the most difficult part of this tool is in estimating how many cycles each task will take per knob value.

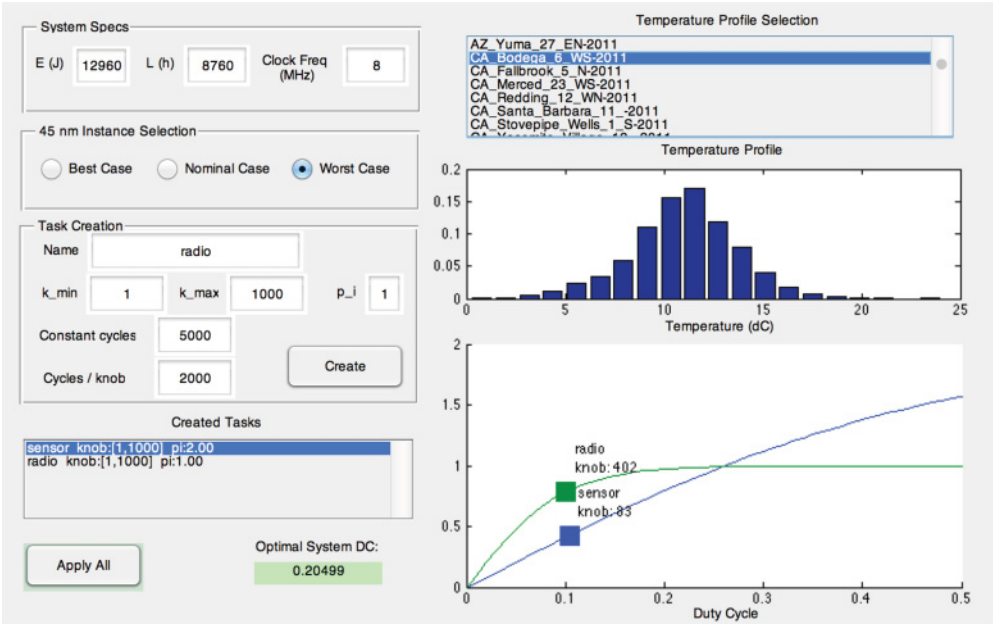

Fig. 10. A tool for guiding developers using VaRTOS. Users input various system specifications along with task prototypes and a geographical location, and the corresponding optimal duty cycle d^*_{sys} and knobs k^*_i are displayed.

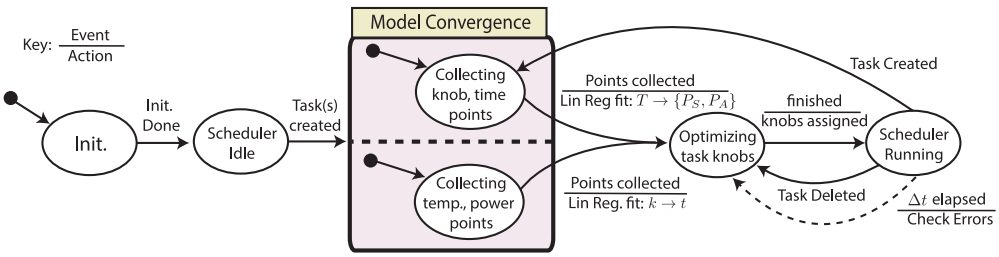

Fig. 11. VaRTOS state chart, showing model convergence and optimization states.

This tool only gives a rough estimate of how the true deployment will behave, but it helps guide the developer's choices along the way.

5.6. Operation

In this section, we discuss the operation of VaRTOS from a broader perspective, using the state chart depicted in Figure 11. 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 every t_{super} seconds (see Section 5.2) to construct $\mathcal{K}_i$. The optimization routine cannot complete until both models have converged, after which linear regression and linearization are used to fit the knob-to-duty-cycle and temperature-to-power curves, respectively. This brings us to the optimization state. Here the various d^*_i are calculated as per Algorithm 1, and the corresponding knob values (calculated by inverting $\mathcal{K}_i$) 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 $\mathcal{K}_i$ and reoptimizing) or task deletion (requiring only reoptimization). 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 in Figure 11 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, though we present no results regarding this extension in this article.

6. EXPERIMENTAL SETUP

We implemented VaRTOS as a series of architecture-independent extensions to FreeRTOS [FreeRTOS Project 2013], a popular open-source real-time operating system. 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. For our evaluation, we use the TI Stellaris LM3S6965 port of FreeRTOS. The LM3S6965 is a microcontroller based on an ARM Cortex-M3 core and is representative of the low-power platforms targeted by VaRTOS.

VaRTOS relies on a temperature- and instance-dependent power model to perform its optimizations and requires appropriate sensors from its underlying hardware platform to build this model. Temperature sensors are typically embedded into most sensing platforms. Energy consumption and power in various processor modes may be measured directly (e.g., as in McIntire et al. [2006]) or indirectly estimated from remaining battery capacity (e.g., with a "smart" battery or as in Lachenmann et al. [2007]). Low-cost probes for variability vectors (including aging, frequency, and leakage power) may be embedded into processor cores and exposed to software as digital counters [Chan et al. 2012].

We evaluate VaRTOS with a series of case study applications under 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 [VarEMU Project 2013], a variability-aware virtual machine monitor.

VarEMU is an extension to the QEMU virtual machine monitor [Bellard 2005] 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 in order to sense and adapt to these variations in software. 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 in the virtual machine. Through the use (and dynamic change) of parameters in the power model, users can create virtual machines that feature both static and dynamic variations in power consumption. A software stack for VarEMU provides virtual energy monitors to the operating system and processes. With the exception of the driver that interfaces with the VarEMU energy counters, VaRTOS running in VarEMU is unmodified from its version that runs on physical hardware.

When starting VarEMU, we provide a configuration file with parameters for the power model described in Section 3. For most test cases, we evaluate the system with three instances (nominal, best case, and worst case) as shown in Figure 3. When necessary for the evaluation, further instances are generated according to SPICE simulation results as described in Section 3. We also provide a trace of temperature based on hourly

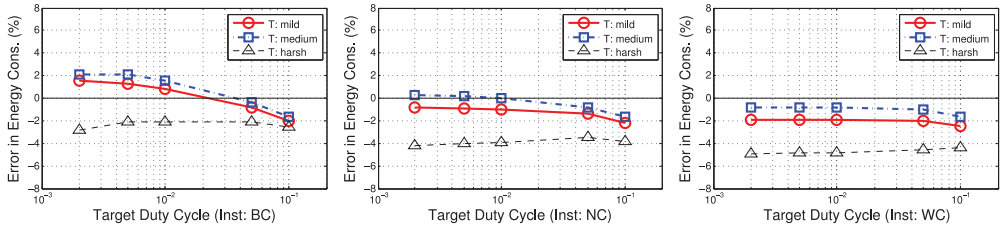

Fig. 12. Error in energy consumption for various optimal duty cycles and for mild, medium, and harsh environments. From left to right, figures represent the best-case, nominal-case, and worst-case power instances in 45nm.

temperature data from the National Climactic Data Center [(USCRN) 2012] 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 1:3,600, resulting in a total simulation time of approximately 2.5 hours for a lifetime of 1 year.

7. EVALUATION

In this section, we present results regarding the ability of VaRTOS to maximize utility by modification of task knob values as well as the corresponding error in energy consumption versus the specified energy budget. Finally, we evaluate the VaRTOS architecture in terms of both energy and memory overheads.

7.1. Minimizing Energy Consumption Error

In order to achieve accurate energy consumption to meet a lifetime goal, VaRTOS needs to be able to accurately achieve the overall system duty cycle d^*_{sys}. To test this, we constructed a simple application with only a single task containing a knob with fine granularity values. Then, using the tool shown in Figure 10, we specified various values for E and L that would ideally lead to a particular d^*_{sys} 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 were $d^*_{sys} \in \{0.002, 0.005, 0.01, 0.05, 0.1\}$, or from 0.2% up to 10%, and the resulting errors in energy consumption are shown in Figure 12. Note that 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 12). 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.

To give more intuition into what this error in energy consumption means, we compared energy consumption for tasks running in VaRTOS (modeling power on a per-instance basis) with those assuming "worst-case" power consumption. True worst-case power consumption is difficult to define, due to the long tail distribution for power across temperature. Because of this, we define worst-case power as the average power consumption for the worst-case instance from Section 6 across the temperature range $[0^\circ C, 45^\circ C]$, particularly $P_s = 330\mu$W and $P_a = 1.187$mW. Figure 13 shows the disparity between the two. Without per-instance power modeling, energy consumption is in some cases over 70% off, and in only one case is it below 10% error. With per-instance power modeling using VaRTOS, the error has dropped to below 2% in most cases and around 5% in the worst case. Note that a positive percent error means a surplus in

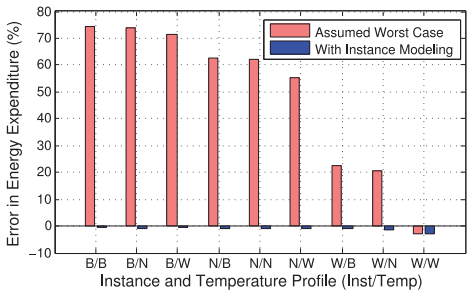

Fig. 13. Errors in energy consumption for a single-task application for worst-case power assumptions and per-instance power modeling.

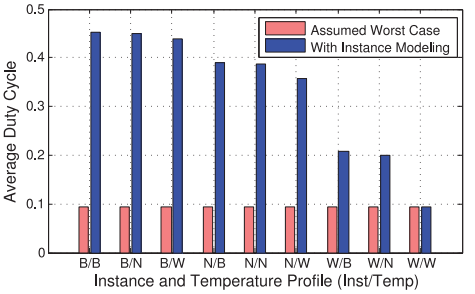

Fig. 14. Average duty cycles of a single-task application for worst-case power assumptions and per-instance power modeling.

energy after the lifetime has been met while a negative means an energy deficit (and likewise a premature death). Energy errors are mostly positive here because of the worst-case power assumption.

Similarly, Figure 14 shows the cause of this energy error disparity—the duty cycle ratio remains constant if worst-case power is assumed while it is allowed to vary when instance-specific power modeling is introduced. This will translate into an increase in the quality of service for a particular application (e.g., more data collected, a higher communication rate) by using what would have been surplus energy.

7.2. Utility and Oracle Comparison

The results in the previous section showed that VaRTOS is able to meet a given energy budget with low error, resulting in an accurate system lifetime. In Section 4, we argued that spending would-be surplus energy will increase system utility. In this section, we substantiate this claim by comparing the utility of the single task app running in VaRTOS to that of an all-knowledgeable oracle system. Unlike the true VaRTOS system, the oracle system is allowed the following privileges: (1) complete knowledge of the temperature profile for the test year, (2) perfect knowledge of task behavior (i.e., $\mathcal{K}_i$), (3) full accuracy models for P_s and P_a, and (4) zero overhead for optimization routines. Figure 15 shows the utilities for both the oracle and VaRTOS. In most cases, VaRTOS achieves within 10% of the oracle utility and is as much as 20% off in the worst case. Note that this comparison is specific to the construction of utility u_i as defined in Equation (5), and other utility curves may cause variations in this metric.

7.3. Energy and Memory Overhead

Energy consumption by the various VaRTOS subsystems must be minimized in order to prevent worsening the very thing we are trying to correct. 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: The amount of program memory (.text, .data) and volatile memory required for VaRTOS depends on the application that the developer is designing. As a baseline, VaRTOS requires a modest increase in the ".text" section over the vanilla FreeRTOS framework from 2.29kB to 6.80kB (a 4.51kB 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 volatile memory, an additional

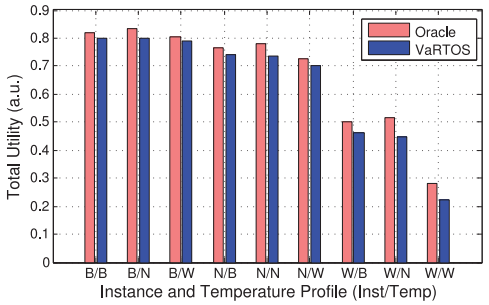

Fig. 15. Total utility, u_{sys}, for VaRTOS versus the oracle system.

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 consumes only 10 additional bytes.

Energy Overhead: The largest energy overhead in VaRTOS comes from the scheduler itself, which, if context swaps occur every 10ms, 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 in the worst case, it consumes $P_{os} = 0.001 \cdot (1.187mW) + 0.999 \cdot (330\mu W) = 331\mu W \approx P_s$. In other words, the scheduler adds only marginal power 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 250μs and occurs only 40 times over the course of a deployment (10ms total); knob perturbations take 48μs and occur $4 \cdot N$ times (for N tasks); performing a 40-point linear regression (for power curves and as an upper bound for modeling $\mathcal{K}_i$) takes 40ms and occurs twice (P_s and P_a) per deployment and once per task; finding d^*_{sys} takes 54ms and occurs once unless tasks are deleted and created after the initial optimization; and finally, finding optimal d^*_i and k^*_i values takes 345μs. In total, these added tasks consume less than 1mJ in the worst case for a 1-year deployment, a negligible overhead if our energy budget is 12,960 joules (two AAA batteries) as in the following section. Note, however, that (1) taking power and temperature measurements is likely to consume additional power for analog-to-digital conversions and (2) a more difficult calculation in energy overhead comes from the result of perturbing knob values in the modeling phase for $\mathcal{K}_i$. The latter depends on the nature and number of tasks, as well as the length of t_{super}.

8. CASE STUDIES

We have shown in Section 7 that VaRTOS can accurately achieve a desired lifetime goal given an energy budget, and we made the claim that using would-be surplus energy will increase application performance. Here we provide several simulated case studies to illustrate this, using the same experimental setup described in Section 6 with additional virtual peripherals as needed. For these case studies, the energy budget is set at a constant 12,960 joules, corresponding roughly to that of two AAA batteries. In addition, the lifetime goal is set at 8,760 hours, or 1 year.

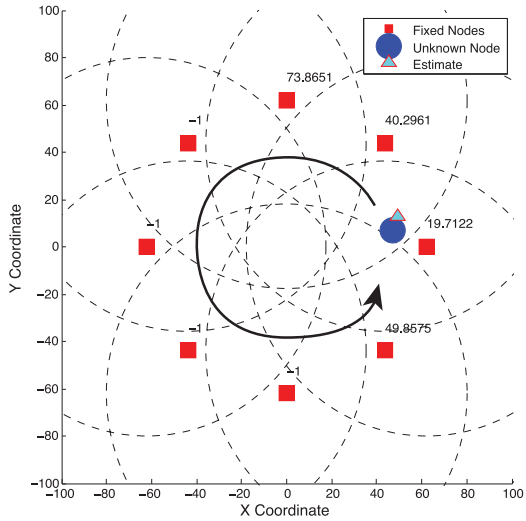

Fig. 16. Multiagent localization application: the square nodes with known location attempt to localize the unknown circle node through consensus.

8.1. Multiagent Applications

Oftentimes, a system is composed of multiple nodes connected by either wired or wireless communication. As an example, consider a network of eight nodes with wireless radio capabilities. Each node is capable of sensing a noisy measurement that corresponds roughly to the distance of some unknown object to the node, whose location is known. For example, these nodes could be taking audio measurements and inferring the distance of a vehicle traveling around a track. From these distance measurements, we would like to estimate the (x, y) coordinate of the unknown vehicle. This system is illustrated in Figure 16, where the unknown object makes a counterclockwise path and the known sensor nodes take noisy measurements if the object is within their radius of observation, shown with dashed circles. Here the unknown object is traveling on a track of 300m circumference at the speed of 0.4km/h. The eight known nodes can "observe" a linear distance to the unknown node if it is within 80m. Each node operates two tasks: (1) a radio with a variable-frequency transmission and (2) a sensor that samples a variable number of points and averages the samples. Allowing the radio more active time will reduce the latency in reported estimates of the unknown node, while allowing the sensing task more time will generate more reliable estimates. The task priorities p_i along with the tool described in Section 5.5 would help a developer give preference to one or the other. For the sake of our comparison, we keep the priorities the same and choose knob ranges to allow the sensor to average between 1 and 100 samples and to allow the radio to transmit anywhere from 10Hz to 0.1Hz. Because these peripherals are simulated, each task has been padded with NOP instructions in order to simulate work that an actual system might be doing. If we look at a 1,000-minute time slice of the estimation process as shown in Figures 17 and 18, we see that the variance of the estimation error is much greater if we assume worst-case power and thus average fewer samples and much less if we use VaRTOS with instance-specific power models. When many samples are averaged, the noise is reduced and each estimate is the result of consensus between more reliable measurements. In other words, for the same lifetime specifications, the system using VaRTOS greatly outperforms the system that assumes worst-case power consumption. While the deployment assuming worst-case

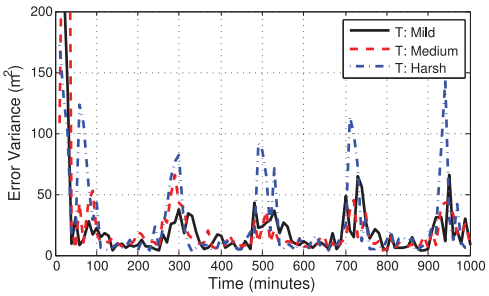

Fig. 17. Reduced error variance for multiagent localization using instance power modeling with VaRTOS.

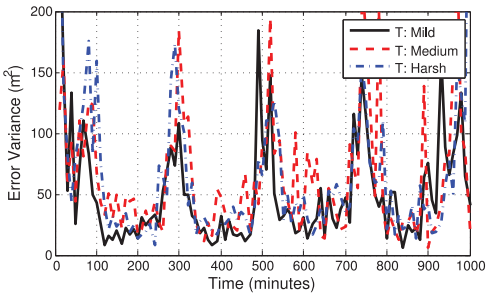

Fig. 18. Increased error variance for multiagent localization assuming worst-case power.

Table II. Task Performance for Multiagent Localization With and Without VaRTOS With a Harsh Temperature Profile

Node ID	#1	#2	#3	#4	#5	#6	#7	#8
VaRTOS # Avgs.	35	34	31	30	28	27	23	10
WC # Avgs.	6	6	6	6	6	6	6	6
VaRTOS Freq (Hz)	1.798	1.719	1.583	1.527	1.459	1.380	1.176	0.525
WC Freq (Hz)	0.287	0.287	0.287	0.287	0.287	0.287	0.287	0.287

consumption suffers from an average error variance of 59.7, the VaRTOS deployment has an average error variance of only 26.9, a 54.9% improvement. The periodic nature of the high-variance peaks in estimation error for both Figures 17 and 18 can be attributed to the circular nature of the application setup (Figure 16). When the unknown object comes within view of nodes with less reliable measurements, the estimation error is much poorer.

Furthermore, the reduction in error variance when using VaRTOS does not come at the price of increased radio latency; the radio latency on the average will improve by using VaRTOS as well, as shown for the case of harsh temperature profiles in the resulting optimal task performance in Table II. The errors in energy consumption for this application are equivalent to those shown in Figure 13, and thus we omit them here for the sake of brevity.

8.2. Prediction-Type Applications

We now move away from wireless sensor network applications and look at systems of just a single node. In particular, we consider an application where we would like to predict one quantity from another correlated but noisy quantity: in our case, we will predict velocity from position, perhaps again on a vehicle of some kind. Here our tool of choice will be the Kalman filter, as it has become such a widely used tool even for resource-constrained applications. We are interested in calculating an estimate $\hat{v}$ of the velocity v from measurements of the position, y, at various frequencies controlled again by a knob. The system evolves according to the simple state space recursion:

$$x_{k+1} = A_k x_k + B_k u_k; \quad y_k = C_k x_k; \quad A_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \quad B_k = [0\ 1] \quad C_k = [1\ 0].$$

For our case, u_k will be a sinusoidal velocity input, and the goal is for $\hat{v}$ to track this input. Intuitively, a faster sample rate for y_k (meaning Δt changes in A as well) should give more accurate predictions of $\hat{v}$, because it is easier to predict states within the near future than it is to predict them in the distant future. The Kalman recursion

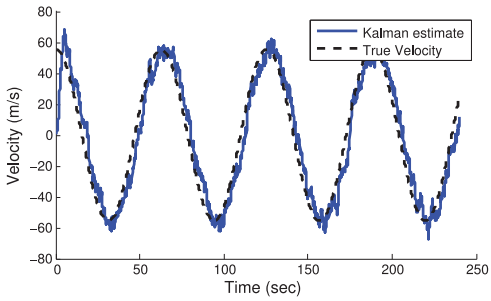

Fig. 19. Kalman filter prediction of velocity from noisy position measurements.

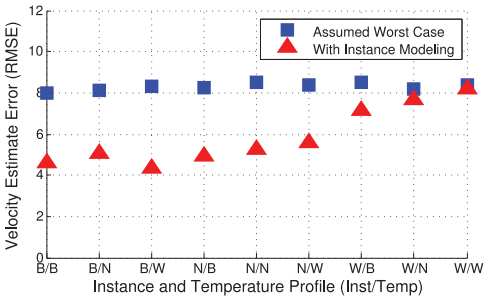

Fig. 20. Error (RMSE) in Kalman predictions when assuming worst-case power consumption and when using VaRTOS.

Table III. Processor Cycle Counts for a Multiblock Signal Processing Application

Block #	1	2	3	4	5
Sensor 1 Block Cycles	20,000	5,000	10,000	25,000	6,000
Sensor 2 Block Cycles	14,000	9,000	15,000	24,000	8,000

itself is omitted here, but we note that the Kalman gain and error covariance matrices (commonly denoted $K_{p,k}$ and $P_{k+1|k}$, respectively) have to be modified on a per-instance basis in order to accommodate the varying sampling period, Δt.

Figure 19 shows an example of the velocity input, u_k, as well as the estimated velocity as calculated by the Kalman filter on position y_k. Here the position readings are subjected to additive white Gaussian noise $\sim\mathcal{N}(0, 50m)$, and likewise the Kalman estimation of velocity contains some noise as well. If we allow Δt to assume values within the range $\Delta t \in [0.1\text{ s}, 10\text{ s}]$ by letting our knob vary as before, the quality of the estimate $\hat{v}$ will increase or decrease in accordance with energy surpluses and deficits, respectively. Figure 20 shows the error (RMSE) in estimating the velocity from noisy position measurements for systems assuming worst-case power and for those using instance-specific modeling with VaRTOS. As before, the x-axis represents combinations of power instances (best, nominal, and worst case) as well as temperature profiles (mild/best, medium/nominal, and harsh/worst). While the worst-case system has a constant error across all combinations, the VaRTOS results show a reduction in prediction error when additional work can be performed without sacrificing lifetime (i.e., those cases where weather and power instance result in a reduction in energy consumption over the worst case). This improvement can be as much as 42.5% in many cases.

8.3. Block Processing Applications

In some cases, a developer may have a number of signal processing (or similar) routines that would help to clean up or extract data from a particular signal. In most cases, the total number of potential sensor processing blocks is likely to be small, and more importantly, it is unlikely that each of these blocks will take the same time and thus the same power to complete. In other words, the functions $\mathcal{K}_i$ that map knob values into duty cycles are no longer a very good fit, as the curve $k_i \rightarrow d_i$ is no longer linear. As an example, consider a system with two sensor streams, each with five potential sensing blocks that all increase the quality of the application as a whole. These tasks each take a distinct number of computer cycles to complete, as summarized in Table III.

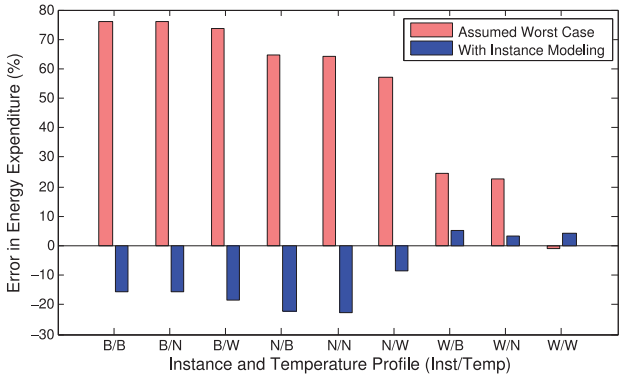

Fig. 21. Energy consumption errors for a multiblock signal processing application.

The cycles here have been arbitrarily chosen in part to show how VaRTOS responds to violations in architecture assumptions (linearity and granularity of task knobs). If we try to schedule these two tasks using VaRTOS with $E = 12{,}960J$ and $L = 8{,}760h$ as before, the resulting errors in energy consumption will be those shown in Figure 21. In this case, the number of blocks executed using VaRTOS is on average 2.8 for sensor 1 and 3.5 for sensor 2, while worst-case power assumption resulted in one block between both sensors combined. In other words, while VaRTOS gives over a 64% signal processing improvement in this application, the errors in energy expenditure are on average much larger than the errors shown in Figure 13; indeed, they would be even larger had our cycle counts shown in Table III varied even more or had the number of blocks been even fewer than five. While VaRTOS can help reduce energy errors to some extent in applications similar to this multiblock signal processing example, care should be taken to ensure that the assumptions made in Section 4 are not completely ignored.

9. CONCLUSION

We developed an architecture for maximizing application quality while meeting lifetime and energy constraints in the face of power and temperature variation. We achieve this through application elasticity as defined by a *knob*—a tunable variable that increases the quality of a given task at the expense of increased power consumption. This knob serves to shape the utility curve of each task as well as offer a means by which the operating system can control the amount of active time and thus power given to individual tasks. We implemented online per-instance power modeling and task modeling in VaRTOS, a series of kernel extensions to the FreeRTOS operating system. Our simulations using VaRTOS show that we can accurately meet a specified lifetime goal with less than 2% error in most cases and less than 5% error in the worst case, whereas had we assumed worst-case power consumption, errors would range from 5% to over 70%. We further demonstrated the ease with which a developer can adopt the VaRTOS architecture; very minimal user input is required, and the effects of these inputs can be tested using a graphical task modeling tool. Finally, we presented case studies for multinode localization applications using wireless sensor networks, estimation problems using Kalman filtering, and multiblock signal processing applications, illustrating how VaRTOS can increase application quality while maintaining lifetime requirements. Knobs in VaRTOS represent flexible notions of elasticity—we make the assumption in this work that they have a linear relationship with computational time, but in general the only assumption required is that an increase in knob value will

increase utility. In other words, if more advanced modeling techniques are used, we can extend this notion of knobs to adapting many other system parameters, not just task frequency and duration. The VaRTOS architecture allows traditional, nonadaptive tasks to coexist with adaptable tasks, and it is therefore suitable for a large class of applications where a notion of elasticity in quality exists. Finally, all software developed in this project is open source and can be found at https://github.com/nesl/vartos.

ACKNOWLEDGMENT

The authors would like to thank Supriyo Chakraborty for help with the initial problem formalization, Liangzhen Lai for valuable input regarding power variation models, and Puneet Gupta for help with the VarEMU project and additional prior work. This material is based in part on work supported by the NSF under awards CCF-1029030, CNS-0905580, CNS-0910706, and CNS-1143667. 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 the NSF.

REFERENCES

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

W. Baek and T. M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. *Special Interest Group on Programming Languages Notices* 45, 6 (June 2010), 198–209.

B. Balaji, J. McCullough, R. K. Gupta, and Y. Agarwal. 2012. Accurate characterization of the variability in power consumption in modern mobile processors. In *Proceedings of the USENIX Conference on Power-Aware Computing and Systems (HotPower'12)*. 8–8.

L. A. D. Bathen, N. D. Dutt, A. Nicolau, and P. Gupta. 2012. VaMV: Variability-aware memory virtualization. In *Proceedings of the Design, Automation Test in Europe Conference (DATE'12)*. 284–287.

F. Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (ATEC'05)*. 41–41.

K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. 2006. High-performance CMOS variability in the 65-nm regime and beyond. *IBM Journal of Research and Development* 50, 4.5 (2006), 433–449.

S. Bhardwaj, W. Wenping, R. Vattikonda, Yu Cao, and S. Vrudhula. 2006. Predictive modeling of the NBTI effect for reliable design. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC'06)*. 189–192.

S. Bhunia, S. Mukhopadhyay, and K. Roy. 2007. Process variations and process-tolerant design. In *Proceedings of the 20th International Conference on VLSI Design*. 699–704.

S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. 2003. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the Design Automation Conference (DAC'03)*. ACM, 338–342.

Y. Cao, P. Gupta, A. B. Kahng, D. Sylvester, and J. Yang. 2002. Design sensitivities to variability: Extrapolations and assessments in nanometer VLSI. In *Proceedings of the IEEE International ASIC/SOC Conference*. 411–415.

T.-B. Chan, P. Gupta, A. B. Kahng, and L. Lai. 2012. DDRO: A novel performance monitoring methodology based on design-dependent ring oscillators. In *Proceedings of the International Symposium on Quality Electronic Design (ISQED'12)*. 633–640.

S. Chandra, K. Lahiri, A. Raghunathan, and S. Dey. 2009. Variation-tolerant dynamic power management at the system-level. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 9 (2009), 1220–1232.

X. Chen, Y. Wang, Y. Cao, Y. Ma, and H. Yang. 2012. Variation-aware supply voltage assignment for simultaneous power and aging optimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20, 11 (2012), 2143–2147.

H. Cho, L. Leem, and S. Mitra. 2012. ERSA: Error resilient system architecture for probabilistic applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 4 (2012), 546–558.

S. H. Choi, B. C. Paul, and K. Roy. 2004. Novel sizing algorithm for yield improvement under process variation in nanometer technology. In *Proceedings of the Design Automation Conference (DAC'04)*. ACM, 454–459.

A. Datta, S. Bhunia, S. Mukhopadhyay, N. Banerjee, and K. Roy. 2005. Statistical modeling of pipeline delay and design of pipeline under process variation to enhance yield in sub-100nm technologies. In *Proceedings of the Design, Automation and Test in Europe (DATE'05)*, Vol. 2. 926–931.

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. 2010. Within-die variation-aware dynamic-voltage-frequency scaling core mapping and thread hopping for an 80-core processor. In *Proceedings of the IEEE Solid-State Circuits Conference (ISSCC'10)*. 174–175.

D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. 2003. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*. 7–18.

FreeRTOS Project. 2013. FreeRTOS. http://www.freertos.org. (2013).

S. Garg and D. Marculescu. 2007. On the impact of manufacturing process variations on the lifetime of sensor networks. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 203–208.

H. Ghasemzadeh, N. Amini, and M. Sarrafzadeh. 2012. Energy-efficient signal processing in wearable embedded systems: An optimal feature selection approach. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'12)*. ACM, 357–362.

S. Ghosh, S. Bhunia, and K. Roy. 2007. 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 (Nov. 2007), 1947–1956.

J. Gregg and T. W. Chen. 2007. Post silicon power/performance optimization in the presence of process variations using individual well-adaptive body biasing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 15, 3 (March 2007), 366–376.

P. Gupta and A. B. Kahng. 2003. Quantifying error in dynamic power estimation of CMOS circuits. In *Proceedings of the International Symposium on Quality Electronic Design*. 273–278.

ITRS. 2010. The international technology roadmap for semiconductors. http://www.itrs.net.

K. Kang, B. C. Paul, and K. Roy. 2006. Statistical timing analysis using levelized covariance propagation considering systematic and random variations of process parameters. *ACM Transactions on Design Automation of Electronic Systems* 11 (Oct. 2006), 848–879.

V. Khandelwal and A. Srivastava. 2007. Variability-driven formulation for simultaneous gate sizing and post-silicon tunability allocation. In *Proceedings of the International Symposium on Physical Design (ISPD'07)*. ACM, New York, NY, 11–18.

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

A. Lachenmann, P. Marrón, D. Minder, and K. Rothermel. 2007. Meeting lifetime goals with energy levels. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys'07)*. 131–144.

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

T. Matsuda, T. Takeuchi, H. Yoshino, M. Ichien, S. Mikami, H. Kawaguchi, C. Ohta, and M. Yoshimoto. 2006. A power-variation model for sensor node and the impact against life time of wireless sensor networks. In *Proceedings of the International Conference on Communications and Electronics (ICCE'06)*. 106–111.

D. McIntire, K. Ho, B. Yip, A. Singh, W. Wu, and W. J. Kaiser. 2006. The low power energy aware processing (LEAP) embedded networked sensor system. In *Proceedings of the International Conference on Information Processing in Sensor Networks*. 449–457.

K. Meng and R. Joseph. 2006. Process variation aware cache leakage management. In *Proceedings of the 2006 International Symposium on Low Power Electronics and Design (ISLPED'06)*. 262–267.

S. Neiroukh and X. Song. 2005. 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)*.

NLopt Project. 2013. NLopt. http://ab-initio.mit.edu/wiki/index.php/NLopt. (2013).

A. Pant, P. Gupta, and M. van der Schaar. 2012. AppAdapt: Opportunistic application adaptation in presence of hardware variation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20, 11 (2012), 1986–1996.

J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic. 1996. *Digital Integrated Circuits*. Vol. 996. Prentice-Hall.

A. Rahimi, L. Benini, and R. Gupta. 2012. Procedure hopping: A low overhead solution to mitigate variability in shared-L1 processor clusters. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'12)*. 6.

S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. 2009. Apprehending joule thieves with Cinder. In *Proceedings of ACM MobiHeld*. ACM, 49–54.

D. Sylvester, D. Blaauw, and E. Karl. 2006. ElastIC: An adaptive self-healing architecture for unpredictable silicon. *IEEE Design and Test of Computers* 23, 6 (2006), 484–490.

R. Teodorescu and J. Torrellas. 2008. Variation-aware application scheduling and power management for chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'08)*. 363–374.

A. Tiwari, S. R. Sarangi, and J. Torrellas. 2007. ReCycle: Pipeline adaptation to tolerate process variation. *The ACM Special Interest Group on Computer Architecture News* 35, 2 (June 2007), 323–334.

J. Tschanz, J. Kao, S. Narendra, R. Nair, D. Antoniadis, A. Chandrakasan, and V. De. 2002. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'02)*, Vol. 1. 422 –478.

UC Berkeley Device Group. 2013. BSIM. Retrieved from http://www-device.eecs.berkeley.edu/bsim/.

U. S. Climate Reference Network (USCRN). 2012. Hourly Temperature Data. Retrieved from www.ncdc.noaa.gov/crn/.

VarEMU Project. 2013. An Emulation Testbed for Variability-Aware Software. Retrieved from https://github.com/nesl/varemu.

H. J. M. Veendrick. 1984. Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits. *IEEE Journal on Solid-State Circuits* 19, 4 (1984), 468–473.

W. Wang, S. Yang, S. Bhardwaj, R. Vattikonda, S. Vrudhula, T. Liu, and Y. Cao. 2007. The impact of NBTI on the performance of combinational and sequential circuits. In *Proceedings of the Design Automation Conference* 364–369.

L. Wanner, C. Apte, R. Balani, P. Gupta, and M. Srivastava. 2012. Hardware variability-aware duty cycling for embedded sensors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 6 (2012), 1000–1012.

H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. 2002. ECOSystem: Managing energy as a first class operating system resource. *ACM Special Interest Group on Operating Systems* 36, 5 (October 2002), 123–132.

R. Zheng, J. Velamala, V. Reddy, V. Balakrishnan, E. Mintarno, S. Mitra, S. Krishnan, and Yu Cao. 2009. Circuit aging prediction for low-power operation. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC'09)*. 427–430.

Received June 2014; revised April 2014; accepted July 2014