A Control-Theoretic Approach to Dynamic Voltage Scheduling

Ankush Varma, Brinda Ganesh, Mainak Sen,
Suchismita Roy Choudhury, Lakshmi Srinivasan, and Bruce Jacob

Dept. of Electrical & Computer Engineering
University of Maryland at College Park
College Park, MD 20742
http://www.ece.umd.edu/~blj/embedded/

{ankush,blj}@eng.umd.edu

ABSTRACT

The development of energy-conscious embedded and/or mobile systems exposes a trade-off between energy consumption and system performance. Recent microprocessors have incorporated dynamic voltage scaling as a tool that system software can use to explore this trade-off. Developing appropriate heuristics to control this feature is a non-trivial venture; as has been shown in the past, voltage-scaling heuristics that closely track perceived performance requirements do not save much energy, while those that save the most energy tend to do so at the expense of performance—resulting in poor response time, for example. We note that the task of dynamically scaling processor speed and voltage to meet changing performance requirements resembles a classical control-systems problem, and so we apply a bit of control theory to the task in order to define a new voltage-scaling algorithm. We find that, using our nqPID (not quite PID) algorithm, one can improve upon the current best-of-class heuristic—Pering's AVG_N algorithm, based on Govil's AGED_AVERAGES algorithm and Weiser's PAST algorithm—in both energy consumption and performance. The study is execution-based, not trace-based; the voltage-scaling heuristics were integrated into an embedded operating system running on a Motorola M-CORE processor model. The applications studied are all members of the MediaBench benchmark suite.

Categories and Subject Descriptors

C.3[Special-purpose and Application-based systems]Real-time and Embedded Systems.

General Terms

Algorithms, Performance, Design.

Keywords

Low-power, dynamic voltage scaling, PID, nqPID.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Appears in *Proc. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2003)*, October 30 – November 1, 2003, San Jose CA, USA.
Copyright © 2003 ACM 1-58113-399-5/01/0011...$5.00.

1. INTRODUCTION

Battery life (i.e. energy supply and rate of depletion) and execution-time performance are arguably the two chief parameters determining the usability of mobile embedded devices such as PDAs, cellphones, wearables, and handheld/notebook computers. The problem is that the goals of high performance and low energy consumption are at odds with each other: while successive generations of general-purpose microprocessors have realized improved performance levels, they have also become more power-hungry. Users demand higher performance without an accompanying cost in battery life or heat dissipation, but it is not always possible to deliver this. Until Intel's recent emphasis on low power, many mobile computers used less-than-cutting-edge processors because the longer battery life and lower heat dissipation of those processors made them more attractive in mobile environments despite their lower performance levels—for example, P3 notebooks when P4 was the norm for desktops, or handhelds that were MIPS-based rather than Pentium-based.

The demand for extracting good performance while having low energy consumption has caused processor manufacturers to take a closer look at power-management strategies. More and more chips supporting dynamic power-management are rolling out everyday, with one of the more popular mechanisms being dynamic voltage-frequency scaling (called simply "dynamic voltage scaling" or DVS [9]), in which the processor's clock frequency and supply voltage can be changed in tandem by software during the course of operation.

Using such a mechanism, a processor can be set to use the most appropriate performance level at any given moment, spreading bursty traffic out over time and avoiding hurry-up-and-wait scenarios that consume more energy than is truly required for the computation at hand (see Figure 1). As Weiser points out, idle time represents wasted energy, even if the CPU is stopped [16].

Voltage and frequency are scaled together to achieve reductions in energy per computation. Scaling frequency alone is insufficient because, while reducing the clock frequency does reduce a processor's power consumption, a computation's execution time is to a first approximation linearly dependent on clock frequency, and the clock-speed reduction can result in the computation taking more time but using the same total energy. Because power consumption is quadratically dependent on voltage level, scaling the voltage level proportionally along with the clock frequency offers a significant total energy reduction while running a processor at a reduced performance level. Transmeta's Crusoe, AMD's K-6, and Intel's XScale (née Digital StrongARM) and Pentium III & IV are all examples of

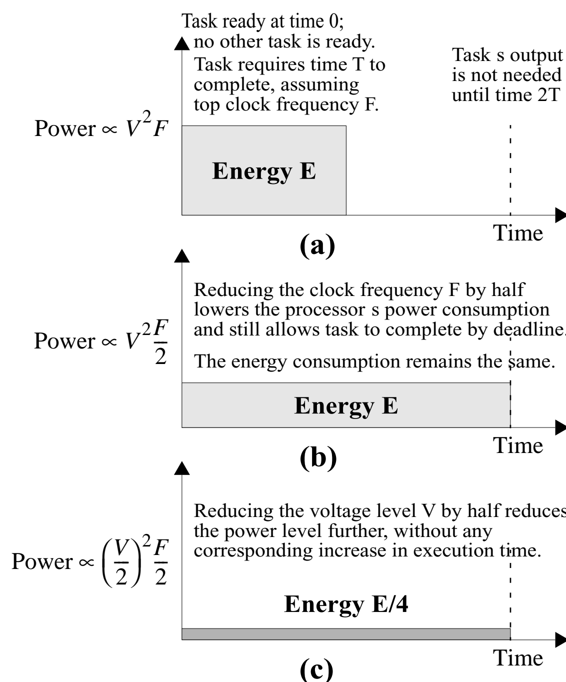

Figure 1: Energy consumption vs. power consumption. Not every task needs the CPU's full computational power. In many cases, for example the processing of video and audio streams, the only performance requirement is that the task meet a deadline, see Fig. (a). Such cases create opportunities to run the CPU at a lower performance level and achieve the same perceived performance while consuming less energy. As Fig. (b) shows, reducing the clock frequency of a processor reduces power consumption but simply spreads a computation out over time, thereby consuming the same total energy as before. As Fig. (c) shows, reducing the voltage level as well as the clock frequency achieves the desired goal of reduced energy consumption and appropriate performance level.

advanced high-performance microprocessors that support dynamic voltage scaling for power management.

However, it is not sufficient to merely have a chip that *supports* voltage scaling. There must exist an entity, whether hardware or software, that decides when to scale the voltage and by how much to scale it. This decision is essentially a prediction of the near-future computational needs of the system and is generally made on the basis of the recent computing requirements of all tasks and threads running at the time. The sole entity that has access to all of this global information (resource usage, demand, and availability) is the operating system [7], because all tasks are visible to it even if the system design makes them invisible to each other. Thus, in actual implementations, the decision to change the processor speed and voltage level is typically made by the OS.

The goal of a voltage-scaling heuristic is to adapt the processor's performance level to match the expected performance requirements. The development of good heuristics is a tricky problem, as pointed out by Weiser et al. [16]: heuristics that closely track performance requirements save little energy, while those that save the most energy tend to do so at the expense of performance—resulting in poor response time, for example.

Recent studies [16, 6, 9, 10, 7, 12] investigate numerous different heuristics that estimate near-future computational load and set the processor's speed and voltage level accordingly. Nearly all of the algorithms studied represent different trade-offs in assigning weights to recent observed processing requirements. The only dynamic variability in the assignment of weights has been through recognizing and exploiting patterns in the performance requirements. For example, a system could predict its future needs from an average of the most recently observed needs unless the recent requirements match some pattern of behavior observed in the distant past, in which case the prediction is based on the distant past behavior instead of the average of recent behavior. However, such schemes have been found to perform no better than static weighted averages [6].

To our knowledge, no published heuristic has incorporated the *rate of change* in a system's processing requirements—the rate of change, the derivative, is a powerful tool at the heart of control-systems theory [3] and well worth exploring in the context of dynamic voltage scaling, as it indicates the rapidity with which a heuristic ought to respond to changes in a system's processing requirements. Without considering the derivative, any heuristic using a fixed assignment of weights will always respond to changes in computing requirements at the same rate. When a relatively idle system is presented with a burst of activity, such a heuristic will take too long to bring the speed of the processor up to the appropriate level, resulting in a window of time during which the energy consumption is low but the system's performance is poor.

This paper presents a heuristic for dynamic voltage scaling that incorporates the rate of change in the recent processing requirements. We took the equation describing a PID controller (*Proportional-Integral-Derivative*), a classical control-systems algorithm [3, 8, 17] as our starting point, and modified it to suit the application at hand — dynamic voltage scaling. We used this new nqPID (not quite PID) algorithm and executed a full system (applications plus embedded OS with the nqPID controller integrated into the OS) running on *SimBed*, an embedded-systems simulation testbed that accurately models the performance and energy consumption of an embedded microprocessor, complete with I/O and timing interrupts, system-level support, and the ability to run the same unmodified binaries that execute on our hardware reference platforms [2].

Most studies of dynamic voltage scaling have been trace-based, with Grunwald et al. [7] providing one of the few execution-based studies. We have implemented the most efficient of the published heuristics, as determined by Grunwald (Pering's AVG_N algorithm [9], based on Govil's AGED_AVERAGES algorithm [6] and Weiser's PAST algorithm [16]), and compared its performance and energy behavior with our scheme. We find that the nqPID scheme reduces CPU energy of an embedded system between 70 and 80% [1] (an improvement over AVG_N of 10–50%), while maintaining real-time behavior: specifically, the jitter in task execution time is limited to roughly 2% (an improvement over AVG_N of a factor of two).

An interesting point is that the algorithm's effectiveness is not obtained through careful tuning of parameters; our results are for a set of controller parameters that were chosen to be conservative. Fine-tuning of the parameters can yield jitter improvements of a factor of two and further energy reductions of 30%.

A sensitivity analysis shows that neither system performance nor energy consumption is particularly sensitive to the nqPID controller's parameters. The jitter values are under 4% of the desired period for all parameter configurations, and more than half of the configurations yield jitter less than 1%. The energy consumption results vary by roughly a factor of two from worst configuration to best: The energy consumed by systems with different nqPID configurations ranges from 17% to 39% of the energy consumed by a system without dynamic voltage scaling. More than half of the configurations yield energy results within 25% of the optimum. We conclude from this experiment that classical control-systems theory is very applicable to dynamic voltage scaling, and we intend to explore the synergy further.

2. BACKGROUND

This section presents brief backgrounds on dynamic voltage scaling, voltage scheduling algorithms, related work, and the structure and behavior of classical PID control algorithms.

2.1. Energy Reduction in CMOS

The instantaneous power consumption of CMOS devices, such as microprocessors, is measured in Watts (W) and, to a first approximation, is directly proportional to V^2F:

$$P \propto V^2 F$$

where V is the voltage supply and F is the clock frequency. The energy consumed by a computation that requires T seconds is measured in Joules (J) and is equal to the integral of the instantaneous power over time T. If the power consumption remains constant over T, the resultant energy drain is simply the product of power and time.

It is possible to save energy by reducing V, F, or both. In practice, V is not scaled by itself. For high-speed digital CMOS, the maximum clock frequency is limited by the following relation [1]:

$$F_{MAX} \propto \frac{(V - V_{THN})^2}{V}$$

where V_{THN} is the threshold voltage. The threshold voltage must be large enough to overcome noise in the circuit, so the right hand side in practice ends up being a constant proportion of V, for given technology characteristics. Usually microprocessors operate at the lowest voltage level that will support the desired clock frequency, so there is not much headroom to lower the voltage level by itself.

The term *voltage scaling* refers to changing the voltage and the frequency together, typically in proportional amounts. The term *voltage scheduling* refers to operating-system scheduling policies that use a processor's voltage scaling facility to achieve higher levels of energy efficiency. For example, it is more efficient to run a processor at a continuous lower speed than to run it at full speed until the task is completed and then idling, as is illustrated in Figure 1.

2.2. Clock/Voltage Scheduling Algorithms

Voltage scheduling can be separated into two tasks [6]:

- *Load Prediction*: predicting the future system load based on past behavior.
- *Speed-Setting*: Using the load prediction to set the voltage level and clock frequency.

The schedulers we have implemented are *interval schedulers* [16]. They perform the load prediction and speed-setting tasks at regular intervals as the system runs. Interval schedulers use the global system state to calculate how busy the system is and typically do not use knowledge of individual task threads.

The primary trade-off is between energy and performance. Weiser observed that a system attempting to run at a flat optimal speed will have significant energy savings, but it will do so at the cost of missing deadlines. On the other hand, a system that responds quickly to changes in the workload will not be as energy-optimal [16].

In the choice of algorithm, this becomes a choice between *smoothing* and *prediction* [6]. A policy that tries to smooth out changes in speed will reduce energy cost, while one that does accurate prediction will improve performance. In theory, it should be possible to achieve both goals to a reasonable degree.

2.3. Related Work

Weiser's seminal 1994 paper on voltage scheduling [16] described the PAST algorithm, along with two other ideal (unimplementable) algorithms. The PAST heuristic predicts that the current interval will be as busy as the immediately preceding interval. To date, it is still one of the best-performing algorithms, as shown by Grunwald [7].

Simulation-based and execution-based analysis of various voltage scaling strategies has been done for workstations [6, 16, 5] and, more recently, low-power embedded devices [12] and PDAs [7, 10]. Govil et al. study a wide range of heuristics from weighted averages of past behavior to pattern matching of processor utilization [6]. Pering et al. look at another range of heuristics and use a "clipped-delay" metric (inspired by Endo et al. [4]) to quantify the resilience of an application to slight degradations in performance [9]. Whereas other studies have used processor idle time as the primary means of predicting future performance requirements, Flautner et al. propose a heuristic that observes inter-process communication patterns to identify scenarios in which the OS can delay tasks until their results are used by another process [5].

Most studies have been trace-driven and/or have considered ideal, oracle-based heuristics to ascertain the optimal energy-performance trade-off. Execution-driven studies, in which a real operating system is instrumented with realizable heuristics, include those by Pering [11], Grunwald [7], and Pillai [12]. Pering investigates design options for extremely low-power general-purpose microprocessors; Grunwald implements the best-of-class voltage-scheduling heuristics in a Linux-based PDA environment; and Pillai simulates an extensive range of parameters, then implements heuristics using a chosen parameter set on an AMD K6-2-based laptop.

1. The resultant energy consumption of the dynamic voltage scaled system is 20–30% of the energy consumed by a system without dynamic voltage scaling.

As far as heuristics go, the cutting edge so far seems to be Pering's AVG_N algorithm, as demonstrated by Grunwald comparing a number of different algorithms on the Itsy handheld computer [7]. AVG_N is a subset and simplification of Govil's AGED_AVERAGES algorithm and reminiscent of Weiser's PAST algorithm. Under AVG_N, an exponential moving average with decay N of the previous intervals is used. That is, at each interval, it computes a "weighted utilization" at time t,

$$[t] = \frac{N \times W[t-1] + U[t-1]}{N+1}$$

which is a function of the utilization of the previous interval U[t]. The AGED_AVERAGES algorithm allows any geometrically decaying factor, not just N/N+1.

Grunwald's findings indicate that AVG_N cannot settle on a clock speed that maximizes CPU utilization. The set of parameters chosen could result in optimal performance for a single application, but these tuned parameters need not work for other applications. However, they found that the AVG_N policy resulted in both the most responsive system behavior and the most significant energy reduction of all the policies they examined.

2.4. PID Control

A PID (Proportional-Integral-Derivative) controller [17] is often used in control systems to make the value of one variable "track", or follow, the value of another with minimum error. In other words, the controller ensures that when the controlling variable changes its value, the controlled variable changes accordingly. Usually, this involves keeping track of past values of both controlling and controlled variables. A full PID control algorithm with feedback has the following equation:

$$y(t) = K_P x + K_I \int_0^t (x - y)dt + K_D \frac{dx}{dt}$$

in which y is the output of the controller at time t , and x is the input at time t. In our case, the input is the measured workload and the output is the estimated workload.

- The ***Proportional*** part of the equation (the first term) makes sure the system reacts as soon as there is a change in the input, and the change in new output tries to follow the input.
- The ***Integral*** part of the equation (the second term) is referred to as the error term or the feedback term, since it measures the net difference between the output an the input so far. It provides the stability in the circuit by making sure that the output follows the input, keeping the system stable.
- The ***Derivative*** part of the equation (the third term) makes sure that the system responds to steady changes in the input efficiently. A rising input corresponds to a positive derivative term, causing the output to rise too.

Simpler control schemes often use just PI or PD controllers for efficient control. However, the PID controller is the one used where efficiency, stability and performance are all required. We show in this paper how an adaptation of this kind of function can be used for efficient voltage scaling in today's microprocessors.

3. METHODOLOGY

3.1. The nqPID Function

Equation gives the equation for a full-blown continuous-time PID controller. We need to simplify this equation and tailor it to the task of voltage scaling before we can implement it as a software algorithm executing on a discrete-time digital computer. We make the following changes:

- We convert the equation from continuous-time to discrete-time, replacing the integral with a summation.
- Similarly, we replace the derivative with the difference between the current and the previous value of x.
- We remove the term y from the right hand side and thereby remove the feedback loop. Systems without feedback have simpler behavior than systems with feedback[2]. Because our goal was to perform a first-order exploration of control-systems theory to the task of dynamic voltage scaling, we felt this an appropriate step. Systems with feedback loops are much more complex than systems without, but they often provide better performance. In future work, we intend to delve into feedback-based algorithms.
- We cut the summation in the remaining integral term (now an average) from an infinite series of terms to a finite series of terms. It now represents the average value of x over the past m intervals.
- We define *utilization* (also called the *system load*) as the fraction of cycles that are busy in a given interval, and *workload* as the product of utilization and CPU speed.

Applying these changes to Equation yields the following discrete-time equation, where the y terms are the predicted loads and the x terms are the measured workloads:

$$y_{n+1} = K_P \times x_n + K_I \times \sum_{n-m+1}^{n} \frac{x_i}{m} + K_D \times (x_n - x_{n-1})$$

This equation represents a function that uses the previous m values of the workload to predict what the next value of the workload will be. This estimate of the workload is used to set the processor's speed and voltage level for the next time quantum. Throughout this paper, "nqPID" will refer voltage-scaling strategies based on this equation. The pseudo-code for the simplified, discrete-time algorithm is given in Figure 2.

We revisit the different terms of the equation to see how this simplified algorithm compares to the full PID algorithm of Equation :

- The ***Proportional*** part of the equation (the first term) predicts that the next value of the load will be the same as that seen the last time. It makes sure that the system can react quickly to changes in workload. If the workload changes suddenly, the operating system can react to it appropriately in the next interval. This is similar to Weiser et. al.'s PAST algorithm [16].
- The ***Integral*** part of the equation (the second term) predicts that the next workload will be the same as the average workload measured in the past few intervals. By averaging and "smoothing the ripples" in the workload, it tries to run the system at a constant optimal speed, thus reducing energy.
- The ***Derivative*** part of the equation (the third term) predicts that if the workload increased in the last interval, it is likely to

2. For example, finite impulse response (FIR) filters are simpler to analyze than infinite impulse response (IIR) filters. In an IIR system, a pulse's effects will never die due to the feedback loop; transient input signals will continue to affect the output forever. To make analysis more tractable, FIR systems put a finite window of time on the effects of any given input signal.

```
// window of previous N actual system loads
load_t load[WINDOW_SIZE];

// Proportional term
estimated_load = Kp * load[0];

// Integral term
for (i=0;, tmp=0; i<WINDOW_SIZE; i++) {
        tmp += load[i] / WINDOW_SIZE;
}
estimated_load += Ki * tmp;

// Derivative term
estimated_load += Kd * (load[0] - load[1]);

// Speed Setting
setPercentSpeed = estimated_load / MAX_LOAD;
```

The proportional term attempts to keep up with changes in the measured system load.

The integral term attempts to average out noise in the system load and thereby avoid switching clock speeds and voltage levels unnecessarily for example, as would happen in response to a brief spike of processor activity during an otherwise idle period.

The derivative term attempts to identify rapid changes in system load so as to keep up when the system suddenly needs high levels of performance, but also to quickly scale performance levels down when the system becomes idle.

Figure 2: Psuedo-code for the nqPID algorithm. The code shows only the speed-setting portion of the algorithm; as a result of the calculations, the CPU's speed is set to be directly proportional to the estimated load. Not shown is the update of the load[] array, any error-checking, etc.

increase again and at the same rate. This is the real predictive part of the equation, because it *anticipates* the changes that might occur in the workload and lets the operating system make a better choice of the required speed setting.

By themselves, these are not very efficient. For example, the proportional part does not adequately study past behavior, and so cannot optimize power requirements. The integral part optimizes energy at the cost of performance by not reacting fast enough (as it is an average of n values). The derivative part by itself cannot predict the actual workload, only changes in the workload, so it is unsuitable for steady-state operation. However, the *consensus* of all of these can provide a very good estimate of what the next value of the workload is likely to be. We take all three effects into account by taking a weighted sum of them.

There are several different but equivalent ways of looking at the nqPID equation:

- From the mathematical point of view, this is a function that *extrapolates* the value of a given variable based on its previous values.
- From the control-systems point of view, this kind of transfer function provides fast response times and low errors.
- From the computer engineering point of view, this is just a *data value prediction* algorithm that is predicting the value of the system load based on its previous behavior.

Another advantage of this type of control is that it is relatively insensitive to changes in its parameters: it provides good response for an entire range, rather than a few values, of the coefficients. In other words, the same coefficients could be used across all applications, while still giving good performance. This is quantified later in our sensitivity analysis.

3.2. System Issues

For this project, Motorola's 32-bit M-CORE architecture [14, 13, 15] was used as the model architecture for our emulator. This architecture was chosen because it is one of the cutting edge embedded processors on the market today, and the M-CORE was designed for high performance and very low power operation. The M-CORE allows 36 voltage scaling levels, corresponding to clock frequencies from 2 to 20MHz.

As the target operating system, we used NOS [2], a bare-bones multi-rate task scheduler reminiscent of typical "roll-your-own" RTOSs found in many commercial embedded systems. We made modifications to the NOS kernel that enable us to choose between a system with no voltage scaling, an nqPID algorithm (with $m=10$, $Kp=0.4$, $Ki=0.2$, $Kd=0.4$) and the AVG_N algorithm (with $N=3$ [9, 10, 7]). The nqPID coefficients were chosen to reflect a middle-of-the-road configuration that would not be fine-tuned to any particular benchmark (cf. Figure 6). At any rate, as the figure shows, the nqPID controller is only weakly sensitive to changes in moderate values of these parameters.

Care was taken to implement the Grunwald implementation faithfully. All parameters other than the algorithms themselves were kept the same to ensure a fair algorithm-to-algorithm comparison. The AVG algorithm was set to change speed whenever the workload drifted out of its optimum range of 50-70% [7].

For benchmarks we used several benchmarks from the MediaBench suite [18]. To assess the effects of an increasing workload, readings were taken with different numbers of tasks running simultaneously (2, 4, and 8), and with tasks having different periods. More than eight tasks could not be run together without completely missing deadlines, and in nearly all cases even 8 tasks represents a workload level at which most deadlines are missed.

Lastly, note that a microprocessor cannot process data while its core voltage or operating frequency is changing. It takes a finite amount of time, and a finite amount of energy, to effect this change. Changing voltage levels and clock frequencies lowers performance and has a "cost" in terms of energy that must be made up by the energy it saves. We use the same time values presented by Grunwald: clock scaling takes 200 microseconds; voltage scaling takes 250 microseconds. The energy consumed during the transition is modelled as if the processor is executing the most energy-intensive instructions.

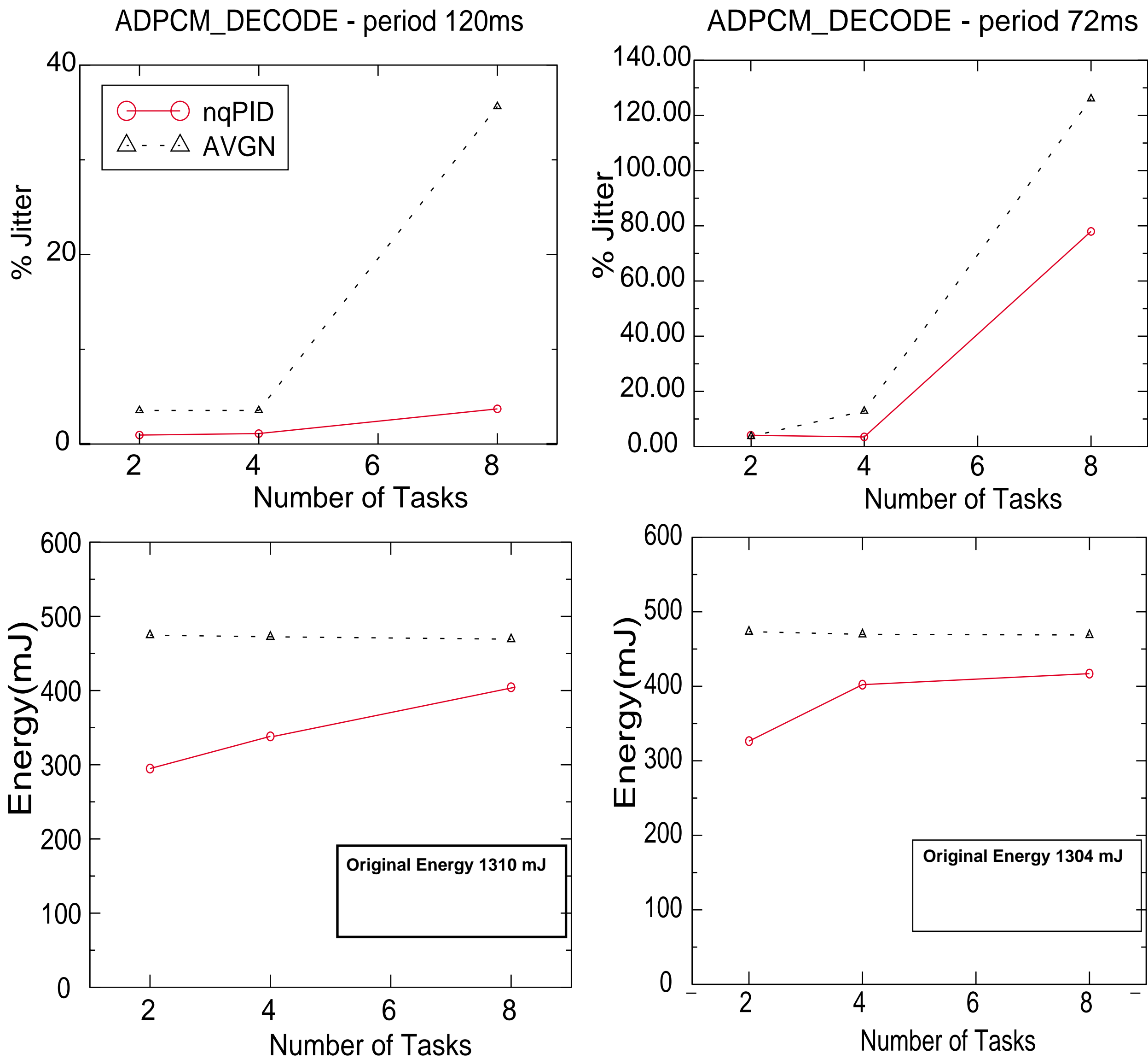

Figure 3: Energy consumption and jitter for ADPCM_DECODE. The graphs on the left represent task periods of 120ms; the graphs on the right represent task periods of 72ms. Energy consumption of the non-DVS-enabled system is reported numerically in the energy graphs.

4. RESULTS

We present a comparison of the two heuristics in terms of how well they trade-off energy and performance; we present a sensitivity analysis; and we discuss how the run-time behavior of the algorithms affects their efficiency.

4.1. Energy and Performance

Our experiments use programs from the MediaBench suite of embedded-systems applications [18] that we have ported to the NOS embedded operating system. To vary the load on the system, two different periods were used for each benchmark, and 2, 4, and 8 tasks were run simultaneously. Each "task" here refers to a producer-consumer pair of processes. To measure the efficiency of the voltage-scaling heuristics, we measure total CPU energy consumed by the system (over a set number of user-level instructions) and the variability in a task's execution time. Energy is reported in milliJoules; jitter is reported as a percentage of the task's desired period. Note that many real-time control systems require accurate timing of task invocations; jitter around a percent or two is perfectly acceptable, while jitter of more than four or five percent can make the operating system completely unusable [17].

Figure 3 compares the efficiency of the nqPID and AVG_N algorithms for the ADPCM_DECODE benchmark. Figure 4 shows

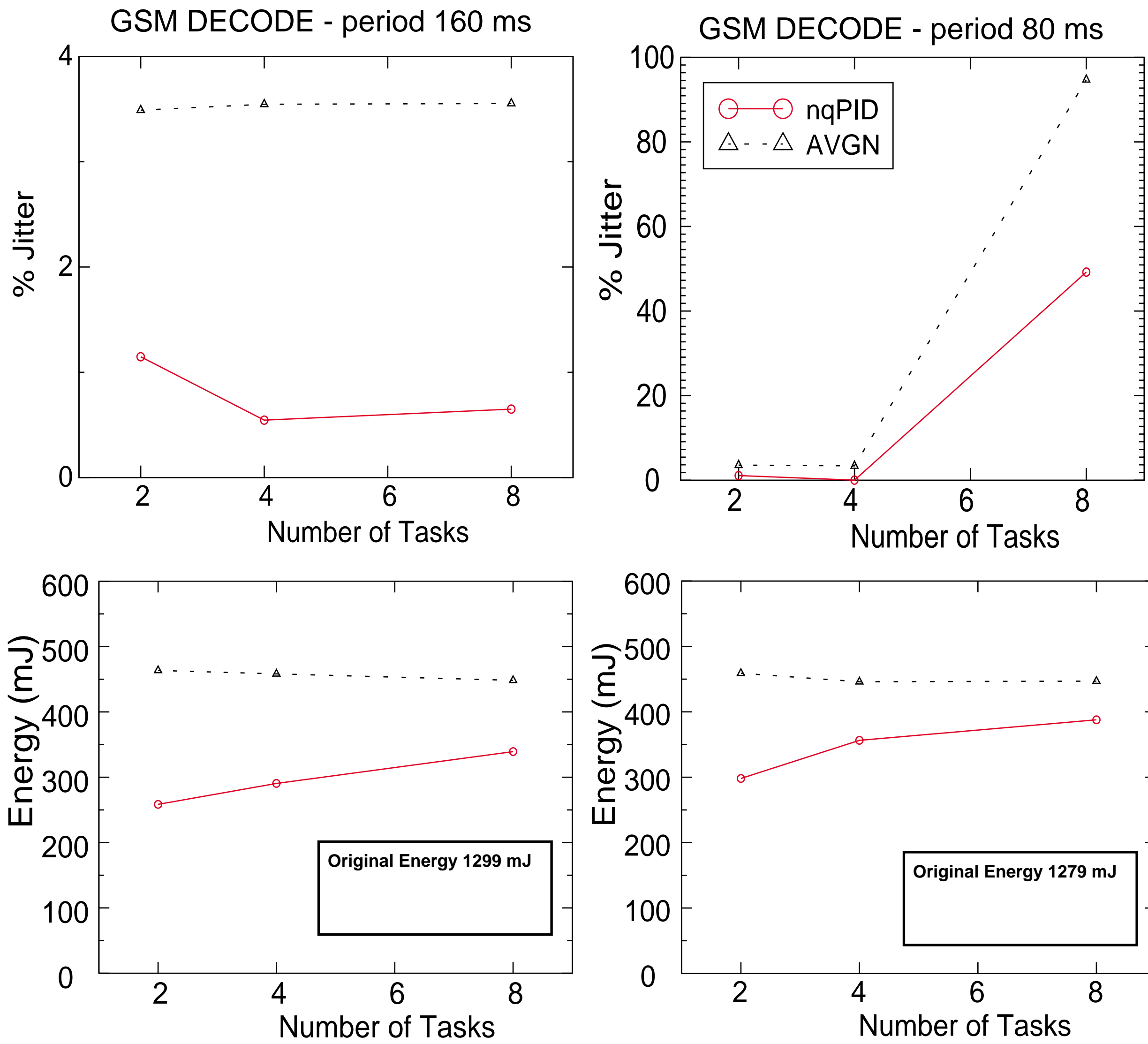

Figure 4: Energy consumption and jitter for GSM_DECODE. The graphs on the left represent task periods of 160ms; the graphs on the right represent task periods of 80ms. Energy of the non-DVS-enabled system is reported numerically in the energy graphs.

results for GSM_DECODE, and Figure 5 shows results for ADPCM_ENCODE.

For each of the figures, the graphs on the left-hand side represent different task periods than those on the right-hand side. For example, in Figure 4, the graphs on the left-hand side represent task periods of 160ms; the graphs on the right-hand side represent task periods of 80ms. Each graph shows results for 2, 4, and 8 simultaneous tasks executing. In all but one case, the workload at 8 tasks represents system overload (the one exception is the 8-task nqPID controller configuration running the 12K-cycle ADPCM_DECODE benchmark, top left of Figure 3 … in this case the jitter is roughly 4%).

We find consistent behavior across all benchmarks and load configurations. The nqPID scheme reduces CPU energy of an embedded system to a point that is 20–30% of what a non-DVS-enabled system would consume. This represents a 75% reduction in energy and is an improvement over AVG_N of 10–50%. The nqPID controller manages to reduce energy to this level while maintaining jitter in task execution time to roughly 2% (for tractable workloads). This represents an improvement over AVG_N of a factor of four.

4.2. Sensitivity Analysis

It can be argued that choosing the nqPID coefficients (the K_P K_I and K_D terms) is difficult to do well and can skew the results if the coefficients are highly tuned for a particular benchmark or set of benchmarks.

Figure 6 shows the sensitivity of the system to choices of coefficients. We chose coefficients so as to sum to 100% (for purposes of the graph, $K_P + K_I + K_D = 100\%$). This represents a planar cut through the design space and simplifies design exploration. This choice is justified so long as analysis shows that the design space is relatively flat, otherwise a larger portion of the space would need to be searched.

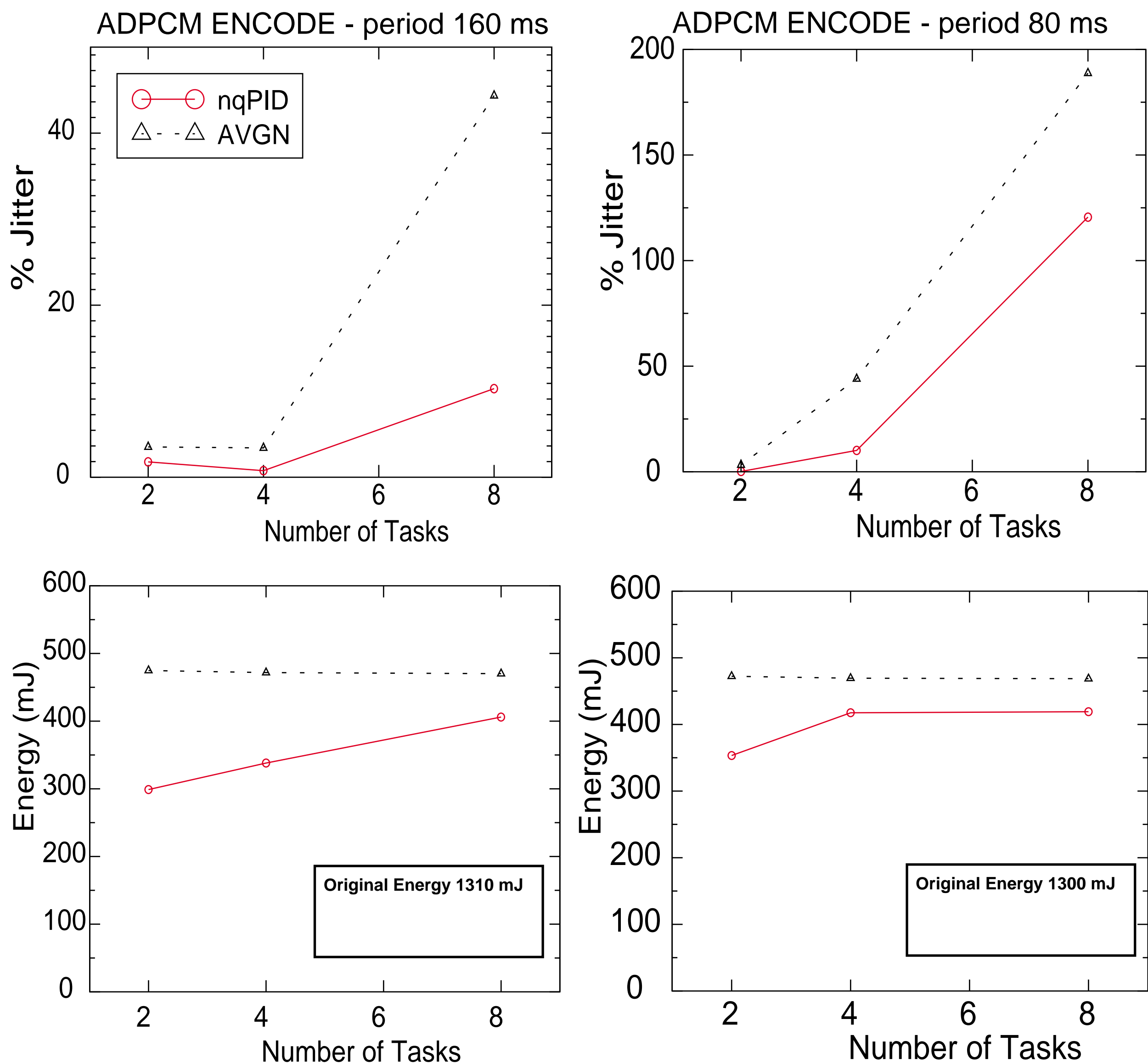

Figure 5: Energy consumption and jitter for ADPCM_ENCODE. The graphs on the left represent task periods of 160ms; the graphs on the right represent task periods of 80ms. Energy of the non-DVS-enabled system is reported numerically in the energy graphs.

Note that the graphs only show different values for K_P and K_I; because we are searching a planar section of the space (not all combinations of Kj are valid), K_D need not be shown as it can be easily calculated. In the figures, circles are drawn around the points corresponding to the coefficients chosen for our experiments: As mentioned earlier, we chose $K_P = 40\%$, $K_I = 20\%$, and $K_D = 40\%$.

The graphs represent the averages over all configurations of all benchmarks—except for 8-task configurations which in most cases represent intractable workloads.

As the graphs show, the design space is relatively flat, and most of the designs lie within 25% of the optimal configuration. For example, in the Energy graph, more than half of the designs lie below the 300mJ mark. In the Jitter graph, we see that the design space is a bit more chaotic, but two-thirds of the designs lie below 1.5% jitter, and no design exceeds 4%. Moreover, there is a large region of the design space, corresponding to values of K_I between 40 and 80 inclusive, that has very flat behavior and excellent jitter values, most under 1%.

Why did we not choose one of these optimal points to begin with, for purposes of comparison? Why did we choose the proportion 40-20-40 for our coefficients? It is often found that coefficients chosen for voltage-scaling studies are extremely benchmark-specific. A particular set of coefficients may run extremely well for a particular benchmark and completely fail to deliver for others. Also, even the values of the coefficients for a particular application cannot be known on an *a priori* basis. We felt that to present the algorithm in the most appropriate light, we must not tailor our parameters to the applications we run, so that our results would indicate how the design would likely fare in a typical real-world implementation. Thus, the proportions chosen are conservative.

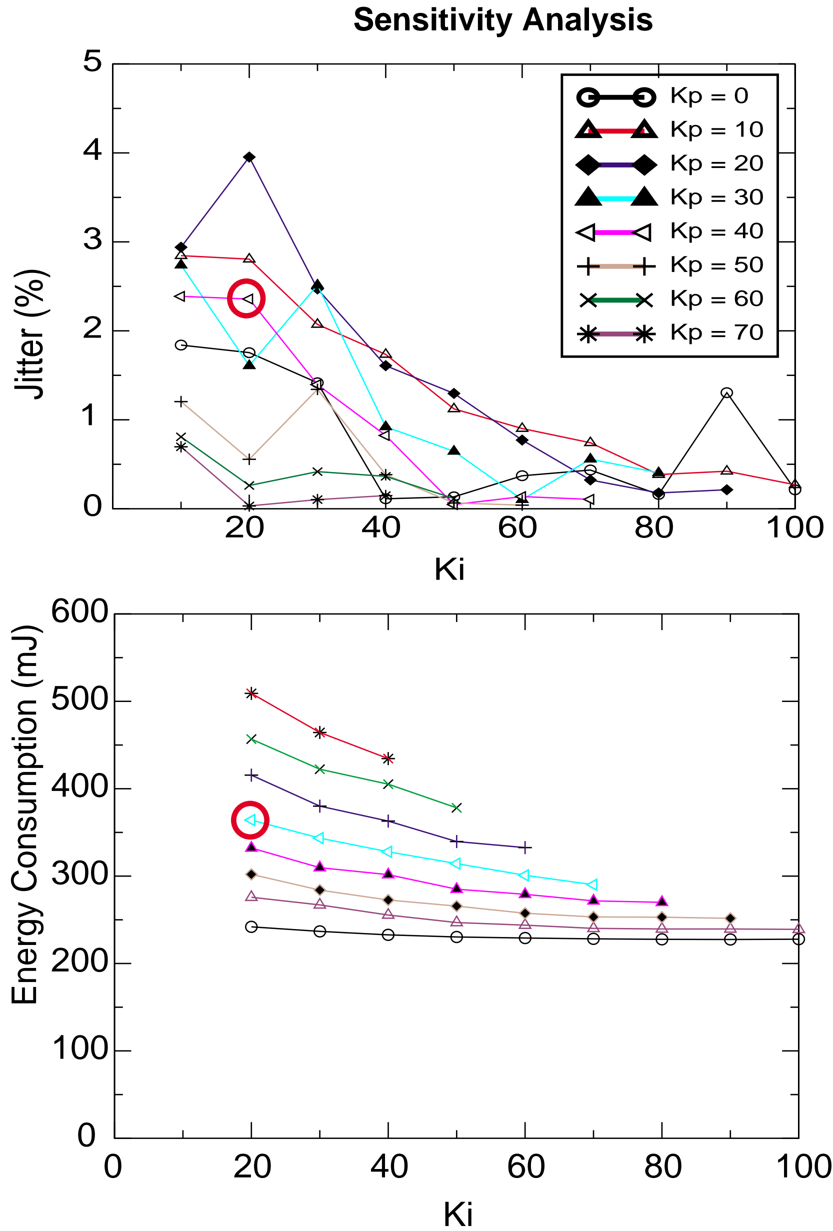

Figure 6: Sensitivity of the system to choices of nqPID coefficients. The coefficients were chosen to add to 100% (Kp + Ki + Kd = 100). Thus, the graphs show different choices for Kp and Ki, as Kd can be easily calculated. Circles are drawn around the points corresponding to the coefficients chosen for our experiments: We chose Kp = 40, Ki = 20, Kd = 40. As the graphs show, the design space is relatively flat.

4.3. Run-Time Traces

For a better understanding of our results, we took snapshots of the systems during execution. Figures 7 and 8 show how the AVG_N and nqPID heuristics respond to system load over a brief window of execution, for two different combinations of benchmark and workload profile. In each graph, the x-axis represents time in µsec. In the top two graphs of each figure, the system load is plotted over time, where 1 is maximum load, 0 is non-loaded. One graph is system load for the operating system using the AVG_N heuristic; the other graph presents system load for the operating system that uses the nqPID heuristic. Note that the two graphs should not be identical, as each heuristic sets the voltage level and CPU speed differently, thereby changing the future system load differently.

In the bottom two graphs of each figure, the CPU speed that the heuristic chose for each time quantum is plotted over time for the same interval shown in the graph above it. Because the voltage level is scaled proportionally with the CPU speed, the CPU-speed graphs also indicate the voltage level at which the processor is running.

Several things are immediately clear when looking at the graphs:

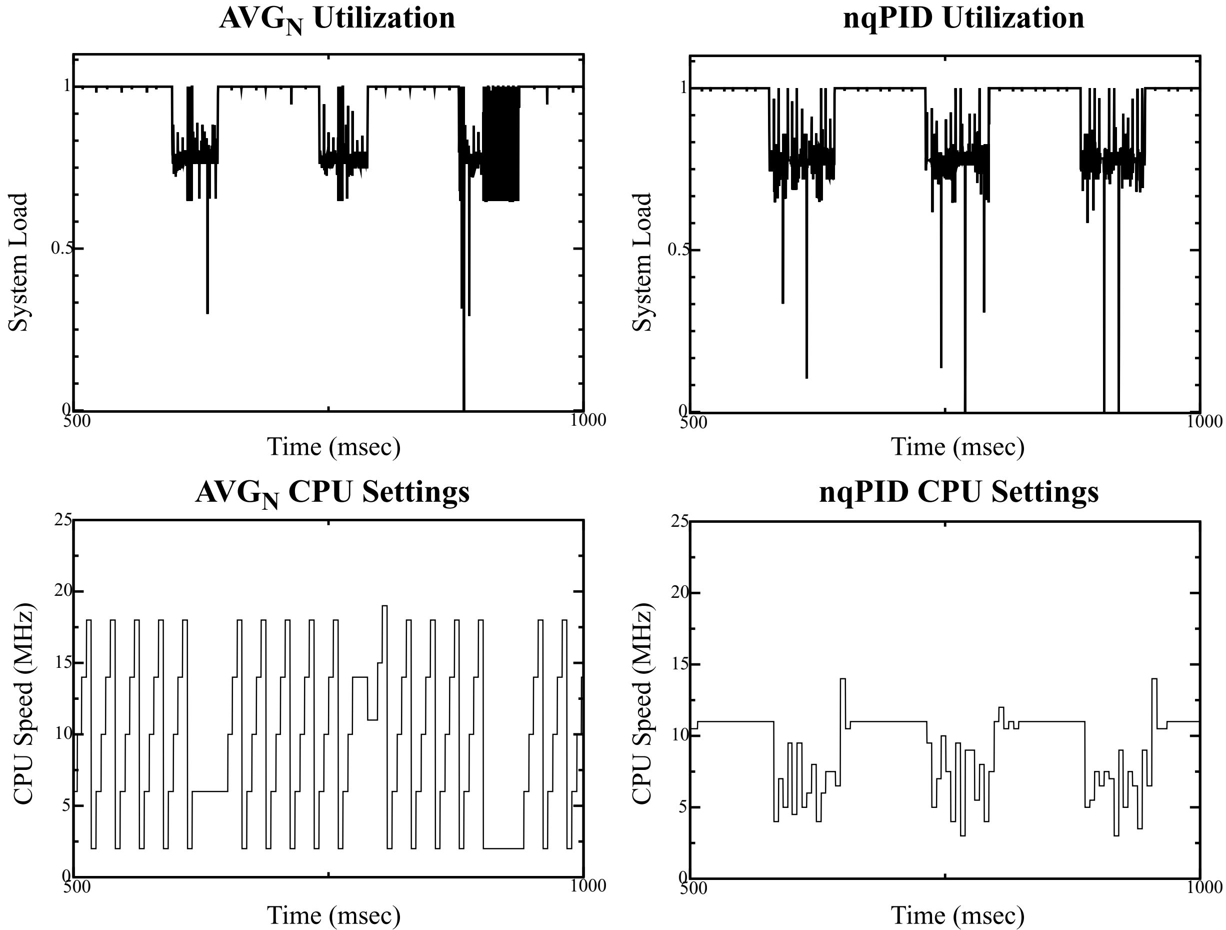

Figure 7: System trace for ADPCM_ENCODE, 4 tasks, 160ms period. The graphs plot the system activity (system load and CPU speed) over the same window of time for the two voltage-scaling heuristics. Time is in milliseconds. CPU speed is in MHz, and system load is between 0 and 1.

- The AVG$_N$ algorithm cannot converge to a solution and keeps oscillating. This behavior of the algorithm is expected, and it was analyzed by Grunwald [7].
- The nqPID algorithm settles down fast as soon as the workload stabilizes (it accurately tracks the workload).
- When the utilization rises sharply and then suddenly stops rising, the nqPID function slightly overshoots it, and sometimes oscillates a little bit before stabilizing. This is as expected. The derivative part of the equation predicts that there will be an upward slope, so this is tracked faithfully. However, it has no way of knowing that the workload will *stop* rising suddenly, hence the overshoot. The slight oscillation before settling down is expected and comes from the control dynamics of the nqPID equation. Intuitively, it is because the when the workload stabilized, the differential part of the equation went to zero, but the integral part of the equation had not caught up yet (because most of the terms in the integral were lower), hence the slight undershoot.
- The nqPID algorithm has less swing in voltage and does not respond to short-duration drops or rises in workload as much. This leads to both energy and performance benefits.

In general, both algorithms make use of the relatively wide range of CPU speed settings available. Weiser points out that a wide range of values can be a dangerous thing, because if a processor is in the low range of performance settings, it might take far too long to get up to an appropriate level when the system load rises rapidly, thereby both reducing performance and wasting energy. However, we have found that the nqPID algorithm, with its ability to track rate of change in system load, seems to respond quickly and appropriately. The overshoot behavior is a hallmark feature of fast response time based on the previously observed rate of change in the system, and one of its great benefits is that it works in both directions: when the system load scales rapidly in both the positive and negative directions, allowing a controller that exploits this information to obtain both appropriate performance when needed and energy savings when system load is light.

5. CONCLUSIONS

Dynamic voltage scaling, or DVS, is a mechanism that enables software to change the operating frequency and voltage level of a microprocessor at run-time and thereby explore trade-offs between performance and energy consumption. Numerous heuristics have been explored with the goal of achieving the best trade-off possible,

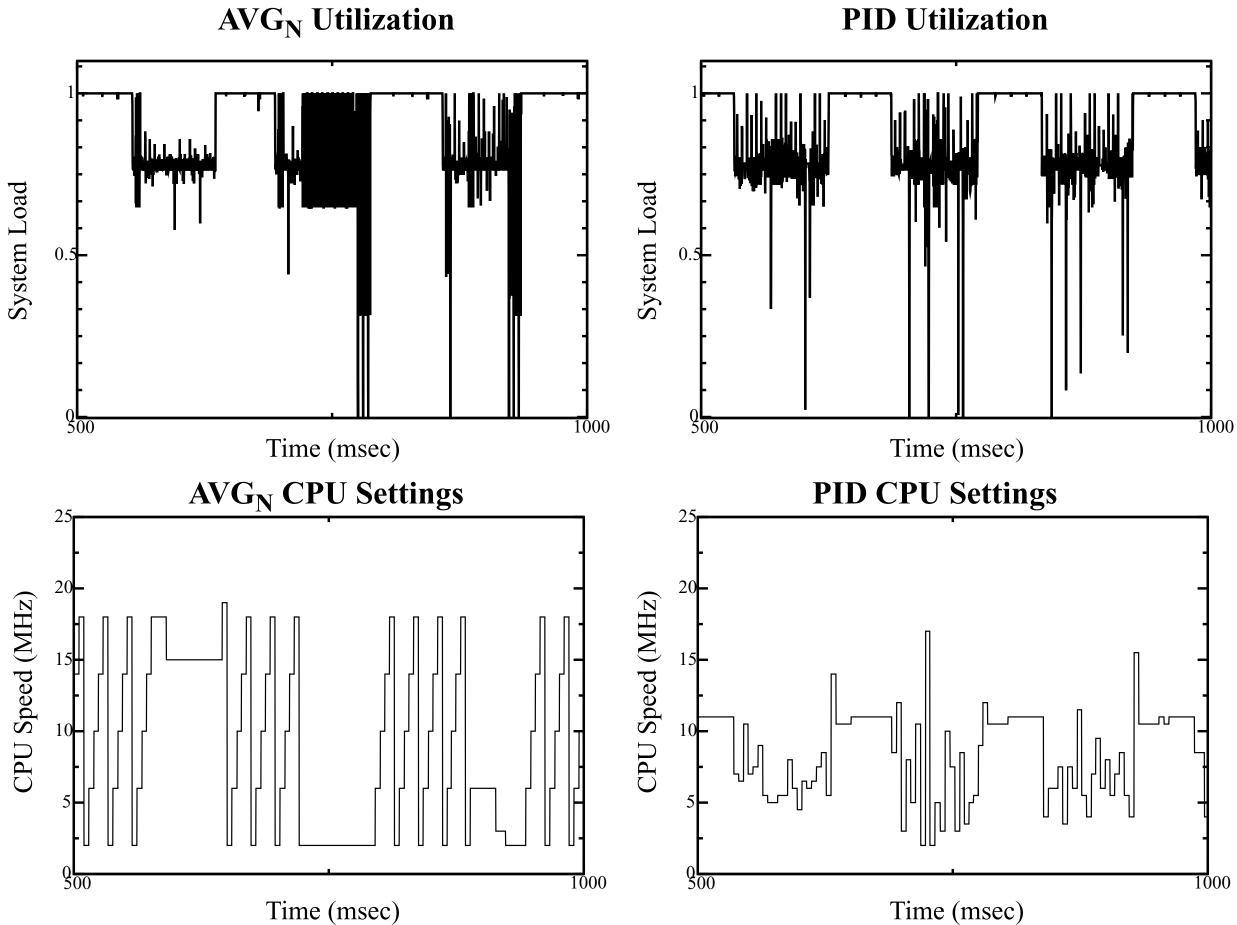

Figure 8: System trace for GSM_DECODE, 4 tasks, 160K period. The graphs plot the system activity (system load and CPU speed) over the same window of time for the two voltage-scaling heuristics. Time is in milliseconds. CPU speed is in MHz, and system load is between 0 and 1.

and nearly all of these heuristics are variations on weighted averages of past system load.

We have developed an nqPID function as a heuristic to control DVS on an embedded microcontroller. The primary strength of our heuristic compared to previous work is that it considers the *rate of change* in the system load when it predicts what the next system load will be.

We implement the controller in an execution-driven environment: a simulation model of Motorola's M-CORE microcontroller that is realistic enough to run the same unmodified operating system and application binaries that run on hardware platforms [2]. The simulation model is instrumented to measure performance as well as energy consumption. The controller is integrated into the embedded operating system that executes a multitasking workload. As a comparison, we also implement the AVG$_N$ heuristic, as it was found by Grunwald to have the best trade-off between energy and performance when implemented on an actual PDA system [7].

We find that the nqPID algorithm reduces energy consumption of the embedded processor significantly. Energy is reduced to roughly one quarter of what it would have been without voltage scaling, and jitter in the system is kept to a small percentage of the tasks' periods. The scheme outperforms AVG$_N$ in both energy consumption and performance as measured by jitter in the periodic task's execution time. Moreover, the algorithm's coefficients were chosen to be very conservative, and a sensitivity analysis shows that the design space is not particularly sensitive to changes in these parameters.

We note that the nqPID algorithm is able to quickly settle to an optimum operating point, whereas a simple running average oscillates about the optimum (this latter feature is not surprising and was already analyzed by Grunwald [7]). In general, control-systems algorithms seem quite promising for voltage scheduling policies.

Our future work will be to investigate more sophisticated control algorithms and to perform a comprehensive comparison and characterization of existing DVS heuristics.

ACKNOWLEDGMENTS

The work of Brinda Ganesh was supported in part by NSF grant EIA-9806645 and NSF grant EIA-0000439. The work of Bruce Jacob was supported in part by NSF CAREER Award CCR-9983618, NSF grant EIA-9806645, NSF grant EIA-0000439, DOD award AFOSR-F496200110374, and by Compaq and IBM.

REFERENCES

[1] R. J. Baker, H. W. Li, and D. E. Boyce. *CMOS: Circuit Design, Layout, and Simulation*. IEEE Press, 1998.

[2] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. “The performance and energy consumption of three embedded real-time operating systems.” In *Proc. Fourth Workshop on Compiler and Architecture Support for Embedded Systems (CASES’01)*, Atlanta GA, November 2001, pp. 203–210.

[3] R. C. Dorf and R. H. Bishop. *Modern Control Systems, 8th Edition*. Addison-Wesley, 1998.

[4] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. “Using latency to evaluate interactive system performance.” In *Proceedings of the 1996 Symposium on Operating System Design and Implementation (OSDI-2)*, October 1996.

[5] K. Flautner, S. Reinhardt, and T. Mudge. “Automatic performance-setting for dynamic voltage scaling.” In *7th Conference on Mobile Computing and Networking (MOBICOM’01)*, Rome, Italy, July 2001.

[6] K. Govil, E. Chan, and H. Wasserman. “Comparing algorithms for dynamic speed-setting of a low-power CPU.” In *Proceedings of The First ACM International Conference on Mobile Computing and Networking*, Berkeley, CA, November 1995.

[7] D. Grunwald, P. Levis, C. B. M. III, M. Neufeld, and K. I. Farkas. “Policies for dynamic clock scheduling.” In *Proc. Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000, pp. 73– 86.

[8] B. C. Kuo. *Automatic Control Systems*. Prentice Hall, 1991.

[9] T. Pering and R. Broderson. “The simulation and evaluation of dynamic voltage scaling algorithms.” In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED’98*, Jun 1998.

[10] T. Pering, T. Burd, and R. Brodersen. “Dynamic voltage scaling and the design of a low-power microprocessor system.” In *Power Driven Microarchitecture Workshop, attached to ISCA98*, 1998.

[11] T. Pering, T. Burd, and R. Broderson. “Voltage scheduling in the lpARM microprocessor system.” In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED’00*, July 2000, pp. 96–101.

[12] P. Pillai and K. G. Shin. “Real-time dynamic voltage scaling for low-power embedded operating systems.” In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP 2001)*, Banff, Alberta, Canada, October 2001, pp. 89–102.

[13] J. Turley. “M.Core shrinks code, power budgets.” *Microprocessor Report*, vol. 11, no. 14, pp. 12–15, October 1997.

[14] J. Turley. “MCore: Does Motorola need another processor family?” *Embedded Systems Programming*, July 1998.

[15] J. Turley. “M.Core for the portable millenium.” *Microprocessor Report*, vol. 12, no. 2, pp. 15–18, February 1998.

[16] M. Weiser, B. Welch, A. Demers, and S. Shenker. “Scheduling for reduced CPU energy.” In *Proc. First USENIX Symposium on Operating Systems Design and Implementation (OSDI’94)*, Monterey, CA, November 1994, pp. 13–23.

[17] T. Wescott. “PID without a PhD.” *Embedded Systems Programming*, October 2000.

[18] C. Lee, M. Potkonjak and W. Mangione-Smith. “MediaBench: A tool for evaluating and synthesizing multimedia and communications systems.” In *Proc. 30th Annual International Symposium on Microarchitecture (MICRO’97)*, Research Triangle Park NC, December 1997, pp. 330–335.