An Effective Feedback-Driven Approach for Energy Saving in Battery Powered Systems

Duy Le
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185, USA
duy@cs.wm.edu

Haining Wang
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185, USA
hnw@cs.wm.edu

***Abstract*—Energy efficiency is essential to battery-powered (BP) mobile systems. However, existing energy efficiency techniques suffer from imbalance between system performance and power consumption. This paper presents a Feedback QoS based Model, called FQM, to successfully achieve power reduction without performance degradation. By observing system behavior via control variables, FQM applies pre-estimated policies to monitor and schedule I/O activities. We implement a prototype of FQM under Linux kernel and evaluate its effectiveness with different applications in terms of power consumption, QoS, and performance. Our experimental results show that FQM can effectively save energy while maintaining high QoS stability.**

***Index Terms*—Energy, QoS, Feedback Control**

I. Introduction

Energy consumption has been a critical issue to battery-powered (BP) computing systems, ranging from laptops to netbooks. The productivity of a BP system heavily depends on its battery runtime. According to [1], with different loads, the runtime of a non-sleep BP system can vary from 0.5 to 5.5 hours. It is a challenging task for these BP systems to achieve high energy efficiency while meeting application quality of service (QoS) requirements [2], [3].

Besides Windows, Linux has become popular for BP systems. As of 2009, Linux is installed in 32% (11 million) of produced netbooks [4]. Under the Linux kernel, new features have been integrated with Intel architectures to save energy [5], including tickless idle and power management (PM) for subsystem components, such as applications, processors, devices, and buses. In particular, Linux kernel version 2.6.23 [6] supports a QoS power management (QoSPM) that enables aggressive power management subsystems without substantially affecting the user QoS expectations. In other words, subject to usability and performance constraints, QoSPM creates the possibility of saving energy from the application level.

In this paper, we exploit this possibility to address a twofold problem: energy saving with QoS provisioning on BP systems. Based on a control theory analysis, we propose a feedback-driven model, called FQM, to optimize interactions between application and system I/O. Specifically, FQM includes five main components: Source, Executor, Monitor, Controller, and Manager. Its input consists of (1) requests that specify application utilization, (2) changes of the feedback utilization that is defined as an FQM metric to specify the system CPU utilization, and (3) references of assigned energy and QoS policies for requests to minimize power consumption. The output includes two controlled variables to be monitored: the miss ratio of those requests that do not meet its assigned QoS, and actual CPU utilization.

The majority of the FQM prototype is implemented under the Linux kernel. Its internal components, located at the kernel level, handle user I/O transactions, assign QoS parameters for produced requests, and monitor the feedback of control variables. The external components, which are initially built from system behavior observations at the user level, are maintained as utilization-based policies. To validate the efficacy of FQM, we conduct a series of real experiments on a laptop running different applications. Our focus is on three aspects of the laptop: power consumption, QoS, and system performance. The power consumption evaluation indicates that FQM can save a significant amount of energy when multiple applications are active simultaneously. The QoS investigation quantifies the variation of QoS parameters and demonstrates the QoS stability in FQM. Finally, the system performance evaluation shows that the overhead induced by FQM is minor, resulting in energy efficiency for the whole system. In summary, FQM can reduce energy consumption by as much as 20% and maintains QoS stability with high utilization and low miss ratio on the running system.

Note that FQM is independent of QoSPM architecture and functionalities, while its prototype works well on QoSPM-based Linux systems. Our FQM design dynamically adapts and guarantees application QoS based on user/system interaction behaviors. This is different from the power-aware systems of Lu et *al.* [7], [8] which do not aim to guarantee the application QoS, the FCS of Lu et *al.* [9], [10], [11], which is applicable only for adaptive real time systems, the EC of Minerick et *al.* [12], which exploits the voltage and frequency of processor to meet a given energy level, the HAPPI of Pettis et *al.* [13], which automatically simplifies power policies but does not considers all primitive I/Os, the Grace-OS of Yuan et *al.* [2], which requires a predictable CPU scheduling and only exploits particular multimedia tasks, and QoSPM, which only statically maintains application QoS.

The remainder of this paper is organized as follows. In Section II, we categorize and specify the IO-based interaction

TABLE I
APPLICATION REQUEST PARAMETER STRUCTURES

FS/NET	Targeted devices (filesystems/networks)
BS	Maximum I/O buffer (bytes)
R/W/S/Re	Primitive IO operations (read/write/send/receive)
Pe/NPe	Transaction source status (periodic/aperiodic)
CS	Total request size (bytes)
DS	Transferred data size (bytes)
Ti	Estimated interval (ms)
To	Estimated time out (ms)

of user application utilization by using requests. In Section III, we detail the design of FQM and its algorithm. In Section IV, we present the implementation of FQM in the Linux kernel. In Section V, we describe experiments conducted on a BP system with and without QoSPM. Finally, we conclude in Section VI.

II. Application Requests

General applications allow users to periodically or aperiodically interact with system I/Os. The I/Os can be considered abstractly as filesystems and sockets, or pragmatically as hard disks and network devices. Inside OS architectures, the power consumption of hard disks and network devices are managed by a subsystem component, called power management (PM). Recently, OnNow [14] and ACPI [15] have been employed as PMs under the kernel level of Windows and Linux, respectively.

From the application perspective, PM is transparent like other subsystem components under the OS kernel. However, it is important to monitor those I/Os that communicate with PM. Instead of directly relying on power states of hardware devices, kernel developers can fulfill the monitoring task via the capability requirements of hardware devices. Specifically, they use such requirements as a power-state based interface between applications and PM. As shown in Table I, an I/O based request (`RQ`) can be considered as an encapsulation of primitive I/O system calls and corresponding parameters.

Each request is associated with a process, which represents application transactions in the system scheduler. Since a running application can hold multiple processes, multiple requests may be produced when an application is performed. Based on the parameters in Table I, the request format is specified as: `RQ=(FS/NET, R/W,PE/NPe,CS,DS,Ti,To)`. Next, we present three examples of application utilization to demonstrate how `RQ` can be employed to monitor user utilization.

A. Online interactions

Online interactions include those applications that allow to exchange data with remote resources. These applications require periodical or aperiodic user interactions to initiate data transmissions over the networks, and then store data in filesystems locally or remotely. For instance, a user opens a webpage, then automatically stores the website content on hard drives or manually on networked filesystems. In this case, the configuration of the data exchanging consists of 15 KBytes of written data with 3 consecutive operations in every 0.2 ms, 10 Bytes of request size, and 1 MBytes of operation buffer. The initial requests to read over the networks, write over the networks, periodical write to filesystems, and aperiodic write to networks are specified as follows:

```
RQ(Pe)=(NET,1MB,R=3,Pe,10B,15KB,0.2ms,-)
RQ(Pe)=(NET,1MB,W=3,Pe,10B,15KB,0.2ms,-)
RQ(Pe)=(FS,1MB,W=3,Pe,10B,15KB,0.2ms,-)
RQ(NPe)=(NET,1MB,W=1,NPe,10B,10KB,.2ms,.3ms)
```

B. Multipart file transmission

A file transmission is a network transaction established between a user and one or multiple hosting sites. Such transactions are based on user interaction with file transmission application. For instance, a user download a file, whose location is specified by a URL. However, a user can customize such a transaction by employing a multipart transmission technique to improve its performance. A single data file is split into segments, which can be simultaneously transferred by several connections. Therefore, on one hand, multiple connections yield an advantage on saturated links, in terms of total bandwidth allocation and resilience. On the other hand, a number of connections used in this technique cause an intensity of NIC operations. Thus, to describe such activities, each transaction needs at least two requests, one for a network interaction to open a data transmission transaction `RQ(R)`, and the other for a filesystem interaction to commit written data `RQ(W)`.

C. Online video streaming

Online video streaming is a standalone application that allows to deliver a video clip to other media receivers. In general, media providers stream video to end hosts. These media are playable by end user media players. These users are referred as the first-hand users. However, such users can also re-stream the received media and deliver it to other destinations, known as the second-hand users. This re-distribution allows the streaming media to be transmitted in another protocol with a coding transition. For instance, TV online streaming providers use RTMP or MMS to deliver their Mpeg-4 encoded media to the first-hand users. However, due to the networks or the decoding process of the second-hand user's media player, only traffic belongs to HTTP or RTP protocols are allowed to receive. Therefore, the first-hand user's stream media player can simultaneously receive incoming stream from the server, transcode, and re-stream it to the second-hand user. These first-hand user activities involve two I/O transactions. Here `RQ(in)` and `RQ(out)` are used to describe the receiving and sending transactions, respectively.

III. FQM - Feedback QoS based Model

The objective of a feedback control (FC) based model is to enhance the user/system interface, and the effectiveness of the model depends on its precision. For general FC models, a three-step procedure provided by Lu et *al.* [11] shows a possibility to guarantee the model precision: (1) specifying correlated parameters, which include desired dynamic behaviors and steady state performance metrics; (2) describing a

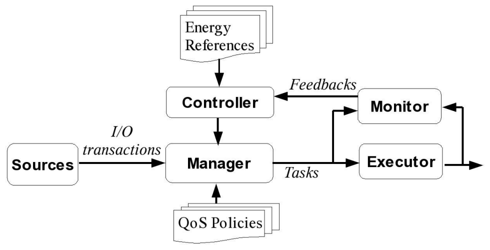


Fig. 1. FQM overview includes internal and external components

relationship between two feature groups—control input and control variables; and (3) designing a steady and transient algorithm with respect to the performance specification and system model. Based on the three steps, we formulate the FQM model and design its algorithm.

A. Model Formulation

Our FQM model includes two groups of components: internal and external. The internal components contain operations that directly involve the feedback loop to monitor data flows. The external components provide information that helps internal ones control data flows. The information from these components can be updated without affecting other operations of the model components. In general, FQM input consists of a set of features, including (1) application requests as control input and (2) feedback utilization modification as control variables. A request describes application utilization, known as a user/application interaction. A feedback utilization modification indicates CPU utilization spent to complete the previous I/O requests of the corresponding transaction. The FQM output includes information to compute control variables. The control variables employed for the feedback loop consist of utilization ratio, miss ratio, and CPU utilization.

To formulate the FQM model, we first specify the attributes of task operation. Here a task $\mathcal{T}$ is the processing of an application request. Since tasks are time based, the attributes are considered at the sampling period i.

•$I_E(i)$: Estimated inter-arrival time between two subsequent task invocations. This attribute is assigned for a task based on the corresponding $\texttt{Ti}_S$, which is referred from a list of $\texttt{RQ}_S$. This list is built based on a CPU cycle pre-estimation with varied I/O operations and QoS level Q. The QoS level is defined based on given a set of QoS parameters, including `BS`, `To`, and `Ti`. Note that these QoS parameters remain intact during the transaction if tasks are periodic.

•$I_A(i)$: Actual average inter-arrival time of task $\mathcal{T}(i)$. The value of $I_A(i)$ is computed based on an accumulation of sampling periods $0\dots(i-1)$.

•$T_E(i)$: Estimated task execution time is known as a time allowed for task completion. This is also assigned for a task based on the corresponding pre-estimated value referred from $\texttt{RQ}_S$.

•$T_A(i)$: Actual task execution time. The value of $T_A(i)$ is the elapsed time on the task before and after its execution.

•$U_E(i)$: Estimated CPU utilization. The value of $U_E(i)$ is computed based on the estimated values of $T_E(i)/I_E(i)$.

TABLE II
UTILIZATION RATIO BASED POLICIES FOR SCHEDULER

	Utilization	CPU	Scheduler
$\delta(i) < \delta_T$	$U_A < U_E$	decreased	`priority++`
$\delta(i) = \delta_T$	$U_A = U_E$	-	-
$\delta(i) > \delta_T$	$U_A > U_E$	increased	`priority--`

•$U_A(i)$: Actual CPU utilization. This value is computed based on actual values of $T_A(i)$ and $I_A(i)$.

•$\delta(i) = U_A(i)/U_E(i)$: Utilization ratio. It represents the difference between actual and estimated utilization values. Specifically, the variation of $\delta(i)$ describes a QoS stability of the current transaction.

•$M(i)$: Miss ratio at the sampling period i^{th} is determined by the number of failed tasks, which do not meet their assigned QoS levels, and the total issued tasks within a time frame $(i-1, i)$. Thus, a task is considered as failure if one of these parameters violates a given condition. For example, a `write` operation of a network-based transaction is failed if the actual delay is greater than its assigned QoS timeout.

1) Internal Components: As shown in Figure 1, five internal components are: Sources, Executor, Monitor, Controller, and Manager. **Sources** initiates I/O transactions. In FQM, this component is known as applications that establish I/Os. In details, Sources initiate and decide whether transactions are periodic or aperiodic. Since multiple transactions can be simultaneously created by different applications, they could instantly produce multiple processes. Consequently, these transactions including multiple processes are delivered to Manager.

Manager processes delivered transactions, including application requests, based on feedbacks. Since simultaneous transactions are specified by their sources, Manager can easily categorize them to create tasks. Basically, Manager handles two major jobs: creating and rescheduling tasks.

First, Manager processes delivered I/O transactions to create tasks. Processes belong to these transactions are identified by IDs, and hence, can be easily differentiated by Manager. Then, Manager creates a task based on process information, which also includes application data. To a structure task, Manager associates the task with assigned QoS parameters referred from a list of $\texttt{RQ}_S$. This list is known as a pre-estimated set of QoS parameters refereed from the external QoS Policies (QP) component. The system functionality requirement is determined by these parameters to execute a task. As a result, task $\mathcal{T}$ created by Manager is structured as: $\mathcal{T}$ = `(ID,[RQ`$_S$`],[Data])`.

Second, Manager uses received control variables to reschedule tasks. These tasks can belong to an incomplete or new transaction. If a transaction is new and has not yet been processed, Manager ignores this task and delivers it to Executor. To minimize CPU utilization on these tasks, Manager must reassign their QoS parameters.

To minimize CPU utilization, we define δ_T as a utilization ratio threshold to estimate the variation of the CPU utilization on a task. Table II lists scheduler policies to adapt the task priority based on evaluation results between δ and δ_T. Since task priorities are dynamically assigned in a given range $(P_{min}\dots P_{Max})$, a different priority will cause a task to be

TABLE III
QoS UTILIZATION BASED POLICIES WHEN $M(i) > M_T$

	CPU	QoS Param
$U_A < U_{RQ}$	decreased	To++ Ti--
$U_A = U_{RQ}$	-	To=To$_S$ Ti=Ti$_S$
$U_A > U_{RQ}$	increased	To-- Ti++

executed at a different CPU level. To reduce CPU cycles spent on a task, `priority` remained its value if provided priority level meets its requirement. Here δ_T is selected as 1 in FQM to show an equivalence expectation between actual and pre-estimated utilization values.

To reassign QoS parameters for tasks, Manager only considers tasks from incomplete transactions. Specifically, for task $\mathcal{T}(i{+}1)$, the QoS parameter reassignment depends on the value of feedback $M(i)$. It should be cleared that QoS parameters to be reassigned include Ti and To. To specify the QoS parameter reassignment, we define U_{RQ} and M_T. $U_{RQ}(i{+}1) = \texttt{Ti}/\texttt{To}$ is defined as a request utilization, which determines an expected CPU consumption for the task $\mathcal{T}(i+1)$. M_T is defined as a possible miss ratio threshold of failed tasks in FQM. Then, the QoS parameter reassignment is specified as follow.

A comparison between $M(i)$ and M_T shows the QoS guarantee of the prior task $\mathcal{T}(i)$. We see that QoS is only guaranteed only when $M(i) \leq M_T$. Thus, Manager does not have to reassign QoS parameters on $\mathcal{T}(i)$. Otherwise, when $M(i) > M_T$, QoS parameters of created task $\mathcal{T}(i)$ must be reassigned. To reassign QoS parameters on a task while moderating its CPU utilization, we only allow Manager gradually readjust the parameters. In detail, such a readjustment is based on utilization based policies, which is shown in Table III. Note that when the actual CPU variation is unchanged, it is similar to the case of $M(i) \leq M_T$. Here, M_T is selected as 0 in FQM to show a requirement that no failed tasks are allowed.

Finally, based on the QoS parameter reassignment at Manager, we formulate (1) as a relationship between control input and control variables, where D and N are numbers of failed and created tasks, respectively.

$$Minimize \begin{cases} M(i) & = \frac{\sum_{j \in D} \mathcal{T}(j)}{\sum_{i \in N} \mathcal{T}(i)} \\ \Delta U_A & = \mid U_A(i) - U_A(i-1) \mid \\ \Delta\delta & = \mid \delta(i) - \delta(i-1) \mid \end{cases}$$

Executor processes tasks delivered from Manager. Since the task structure includes `Data` and `RQ`, Executor must extract the task to get data, then processes the data based on QoS requirements, which are specified in `RQ`. FQM considers the output of Executor as the finish status of completed tasks. Since no failed tasks are allowed in FQM, Executor must completely process tasks even though its holds the lowest QoS level.

Monitor measures FQM attributes, such as $I_A(i)$, $T_A(i)$, and $M(i)$, for the feedback loop to compute control variables by intercepting Executor, specifically, before and after the task execution. In detail, before task is executed, Monitor measures I_A. Other attributes are measured when the execution is completed. Note that, since the source status could be periodic or aperiodic, instead of periodically quantifying such attributes, the measurement of Monitor is active when tasks are started to be processed.

TABLE IV
PRE-ESTIMATED QoS POLICIES

RQ$_{S(11)}$	RQ$_{S(12)}$	...	RQ$_{S(ii)}$	...	RQ$_{S(NN)}$
BS$_1$	BS$_1$	...	BS$_i$	...	BS$_N$
To$-\Delta$To	To	...	To$-\Delta$To	...	To$+\Delta$To

TABLE V
PRE-ESTIMATED ENERGY REFERENCES

DS	FS	Net
S_1	$T_E(FS, S_1, op_1)\ldots$	$T_E(Net, S_1, op_1)\ldots$
...	...	...
S_N	$T_E(FS, S_N, op_1)\ldots$	$T_E(Net, S_N, op_1)\ldots$

Controller receives values delivered from Monitor, then computes control variables based on referred features from Energy References (ER). Specifically, Controller must determine $I_E(i)$ and $T_E(i)$ to compute $\delta(i)$. For $I_E(i)$, Controller can determine this value since $I_E(i) =$Ti and Ti is assigned by Manager for $\mathcal{T}(i)$. For $T_E(i)$, Controller refers this value from ER, the pre-estimated list of CPU cycles, based on the specification of task $\mathcal{T}(i)$, such as targeted device, I/O operation, and source status. Finally, control variables, including $\delta(i)$, $U_A(i)$, and $M(i)$, are feedbacked to Manager.

2) External Components: Manager and Controller refer QoS features from external components QoS Policies (QP) and Energy References (ER) to compute control variables and reassign QoS parameters. Therefore, such features of QP and ER must be available before internal components start to process tasks. Note that these features are updated independently by external components without affecting the feedback loop.

QoS Policies preserve a list of RQ$_S$ categorized by transaction sources. To determine RQ$_S$, which includes Ti, To, and BS, FQM needs to pre-estimate system behaviors. This pre-estimation is established with input as different I/O transactions. Specifically, when these I/O transactions, which include multiple processes, are sequentially performed, QP determines QoS parameters that are associated with each process, to create RQ$_S$. For Ti and and To of RQ$_S$, due to the QoS guarantee requirement for each RQ$_S$, these parameters are determined so that the performance to process this RQ$_S$ does not depend on CPU occupation. Therefore, it is similar to $U_E(\texttt{RQ}_S) = 1$, or Ti = To. For a significant range (To$_{min}$,To$_{Max}$), we define the range as (To$-\Delta$To,To$+\Delta$To), where ΔTo is a related threshold to To. For example, the timeout of a `write` operation in web browsers is set as 2 seconds [16], to select ΔTo as 25%To, we determine a significant range for this timeout as (1.5,2.5). For BS, which determines a size of I/O buffer that caches task before it is processed, the data size DS of RQ$_S$ is accumulated. BS will be increased when this accumulation achieves the current BS value. Note that the initial value of BS determined in RQ$_S$ must cache the minimum data size DS. For example, `send` operation initially requires at least 4 Bytes. Therefore, this value is set for BS as default. Finally, QP is structured as a list illustrated in Table IV.

Algorithm FQM: Feedback QoS based Model

```
Input  : I/O Transactions
Output: Task Execution Results
1  input ← Source initiates I/O Transactions;
2  while (I/O) do // Keep the feedback loop
3      $T_{new}$ = I/O Transaction;
4      $\mathcal{T} \leftarrow T_{new}$; // Get new tasks
5      $\mathcal{T}(j)$ feedbacks received;
6      for $\mathcal{T}(i) \in (T_{new}, T_{old})$ do // All tasks
7          if $\mathcal{T}(i) \in T_{old}$ then
8              Manager($\mathcal{T}(i)$);
9              Evaluate($\delta_{(j-1)}$,$\delta_T$);
10             Assign priority for $\mathcal{T}(i)$;
11             Evaluate(DS);
12             Increase BS if needed;
13             RQ←QoS Policies;
14             Evaluate ($M_{(j-1)}$, $M_T$);
15             Identify RQ$_S$;
16         end
17         AssignQoS(RQ$_S$) → $\mathcal{T}(i)$;
18         Executor($\mathcal{T}(i)$) → output;
19         Monitor($\mathcal{T}(i)$);
20         Controller($T_A(i)$, $I_A(i)$, $U_A(i)$, $M(i)$);
21         Energy References → $\delta(i)$;
22     end
23 end
```

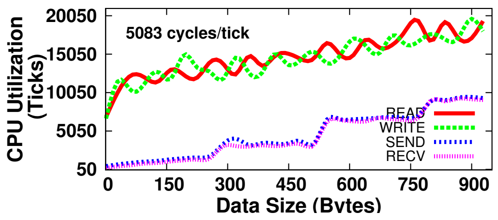


Fig. 2. Average CPU utilization to process primitive operations

Energy References maintain a list of pre-estimated CPU utilization of I/O transactions. These I/O transactions include primitive I/O operations *op*, such as `read`, `write`, `send`, and `recv`. To pre-estimate CPU utilization for these operations, we build a list of CPU cycles from individual processes of emulated tasks. An emulated task is also structured as (`FS/NET`,`[Data]`). When a task is processed, T_E is estimated based on the variation of its data size DS and targeted device. It is clear that the precision of this estimation depends on the fine-grained value S_i of data size DS. As illustrated in Table V, ER is structured as a list, which is categorized by S_i. Here S_0 and S_N are known as the minimum and maximum capacity of the data size DS. Considering CPU behaviors of this pre-estimation, Figure 2 shows the average CPU utilization to process primitive operations. In a 32-bit system, the actual minimum data size of `read` and `write` operations, whose CPU utilizations can be differentiated by FQM, is 16 Bytes. Therefore, the default value of S_i is set to 16 Bytes.

B. FQM Algorithm

The FQM Algorithm embodies a feedback loop to monitor control variables δ, M, and U_A. The algorithm considers input as I/O transactions, which include I/O operations from different sources, and output as process results of these I/O operations. In general, FQM algorithm is constructed by two embedded loops.

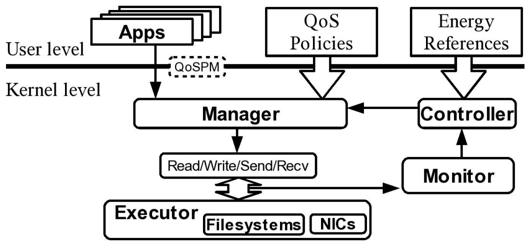


Fig. 3. FQM implementation in Linux-based system

First, the outer loop (lines 2-23) verifies if I/O transactions are still available to be processed. Then, when a transaction is received, Manager will create new tasks based on its process information. Meanwhile, Manager needs to verify delivered feedbacks from Controller, which include control variables (lines 3-5).

Second, the inner loop of this algorithm embodies the feedback loop (lines 6-22). This loop considers the tasks of both new and incomplete transactions. However, Manager only reassigns QoS parameters for those old tasks whose prior tasks in the same transactions are failed. To reassign QoS parameters for these old tasks, Manager needs to evaluate the utilization ratio δ_{j-1} of the prior task and current values of miss ratio M_{j-1} and data size DS. Then, Manager determines a proper value of RQ$_S$ for the reassignment after these evaluations (lines 8-15). After that, the QoS assignment will be established for all old and new tasks, before it is delivered to Executor (lines 17-18). Actual FQM attributes at Executor are measured by Monitor and, later, delivered to Controller to compute control variables (lines 19-20). By referring features of CPU utilizations from ER, Controller computes control variables and $\delta(i)$, which are used by Manager in a new feedback loop (lines 20-21).

IV. Implementation

FQM is implemented in Linux system at the user and kernel levels. As shown in Figure 3, the implementation is independent from QoSPM. In this section, we first describe the implementation of external and internal components in FQM. Then, we discuss the possible impacts of FQM upon the BP system.

A. CPU cycles and QoS parameters

The external components ER and PQ are implemented at the user level, and initially operate as individual tools. These tools monitor system behaviors upon the variations of other application I/Os and create pre-estimated lists of CPU cycles and QoS parameters.

For CPU cycles, ER is performed at the user level to estimate the utilized CPU cycles of primitive kernel operations, such as `read` and `write` for filesystems, and `send` and `recv` for network I/Os. The estimation results are stored in a list as a pre-estimated energy reference, which is used by Controller at the kernel level. However, if Controller refers data in ER via an user/kernel interface, it will cause significant overhead. To avoid this, a copy of the ER list is stored at the kernel memory. In order to guarantee the consistency between two replicas, any modifications of ER list at the user level are immediately updated to its copy at the kernel. For QoS parameters, it is similar to ER. QP initially operates as a module to emulate tasks while estimating QoS parameters. The estimation results of these QoS parameters are stored as a list of QoS policies. A copy of this list is also stored at the kernel level and will be used by Manager. For the accuracy of these pre-estimations, the implementation must consider two factors: overhead and stability.

Regarding incurred overhead, which is due to system calls invocation from the user level, ER and QP use `rdtsc` to obtain the high-resolution CPU timing information with the lowest overhead. This method allows to precisely estimate T_E based on the given CPU configuration and the number of ticks for each operation. Note that final values T_E are computed with an estimated error specified by the sample standard deviation. To maintain the system stability, ER and QP require that subsystem components and any other user applications cannot be performed when primitive operations are individually invoked. This requirement minimizes the interferences induced from other simultaneous I/O operations.

B. Feedback loop

The internal components of FQM are implemented inside the Linux kernel by intercepting kernel functions and system calls. First, for control inputs, Manager differentiates applications and their I/O transactions by using process information. The processing information of a running application can be obtained through the `/proc` file systems. To allow Manager access this information, the application is associated with a given `/proc` filesystem when it is performed, e.g., Firefox is associated with `/proc/firefox`. Manager then creates tasks by intercepting I/O system calls, such as `read`, `write`, `send`, and `recv`. For a multi-process transaction, Manager creates the first task based on the core process, then the other tasks that are associated with other sub processes referred from the system process list. Note that Manager does not distinguish these tasks from others when delivering them to Executor.

Second, for control variables, to obtain their actual values, Monitor also uses `rdtsc` to minimize incurred overhead. After Controller finishes the computation of control variables, these variables are stored in a cache and then are read by Manager. The cache is structured as a circular buffer. The initial buffer size is set to 10,000 elements, and the size is incremented by 1,000 if the number of feedbacks reaches the current limit. As default, the cache is set as write-only for Controller and read-only for Manager.

Finally, with respect to the incurred overhead of searching data in the list of QP and ER, we need to employ an effective searching technique. Since QP and ER are categorized and ordered by given features, such as `BS` in QP and `DS` in ER, using the binary search allows them to quickly locate proper items in the list. Note that the accuracy of this search depends on the match of the search value and the items in the list. For example, if the search value `DS` on the list ER could not be found, the returned result will be the data size S_i, which is closest to `DS`.

TABLE VI
APPLICATIONS AND EXECUTION DETAILS

	Data (MB)	**Transaction**	`read`	`write`	`send`	`recv`
Firefox	35.46	5	62347	73625	1126	3584
Caxel	220.23	6	42599	89398	433	5125
VLC	65.12	3	93101	86831	3231	2385

V. Experimentation

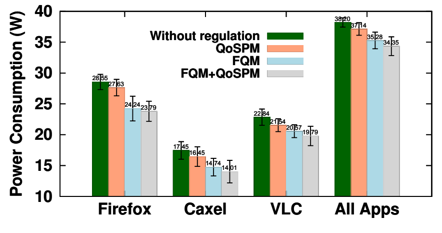
We run user applications on a Linux system to evaluate the prototype of FQM. We focus on three aspects of the system: power consumption, QoS, and performance. Our experimental platform is a Dell laptop with 1.6GHz Intel Pentium M processor running the Linux kernel version 2.6.29 on ext3 filesystems. Table VI lists the details of application executions, which include *Firefox*, *Caxel*, and *VLC*. Note that these applications are associated with given `/proc` file systems, and hence they do not need to be modifed for working with FQM.

We use *Firefox* to exploit user I/O interactions in exchanging data with remote resources. Because the variation of these I/O transactions depends on the web page content and user behavior, we normalize keystroke patterns generated by Autokey [17] to periodically and aperiodically retrieve contents of 50 common websites [18]. We consistently maintain the website list and keystroke patterns for all tests to minimize interferences induced from other workloads.

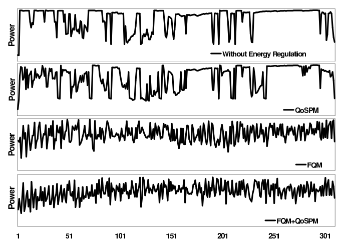
We build *Caxel* based on axel [19], a multipart command-line file transmission, and enhance its transmission features. During the transmission, *Caxel* can modify connection configurations or schedule transmitted files. For the experiments, we use six data files (10-100 MB), which are stored in another machine. That machine accepts up to 50 simultaneous connections for each transmission. Note that FQM considers the source status of *Caxel* as periodic or aperiodic when the transmission is configured as consecutive or scheduled.

We employ *VLC* [20] as a media player and a server streamer. The *VLC* stream server on the 1^{st} machine uses RTP to broadcast three video files (30 MB each, MPEG-1, 1.25 Mbit/s). The *VLC* player is performed on the 2^{nd} machine—the Dell laptop with the FQM prototype installed—to restream this delivered RTP stream in HTTP to the 3^{rd} machine. Another *VLC* player runs on the 3^{rd} machine to receive and validate this delivered HTTP stream.

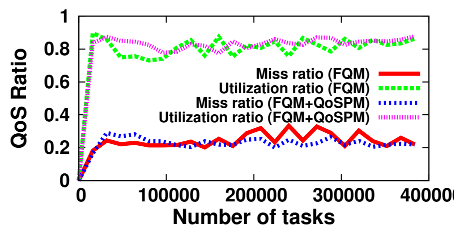
For power consumption, we quantify and analyze power consumption of the BP system in four different configurations: vanilla system (*without energy regulation*), vanilla system with QoSPM activated (*QoSPM-based*), system with FQM installed (*FQM*), and system with QoSPM activated and FQM installed (*FQM+QoSPM*). We use a standalone power meter to measure the system consumed power. For QoS evaluation, we focus on the variation of QoS policies applied on tasks. Finally, the system performance evaluation is mainly focused on the overhead of FQM in vanilla system.



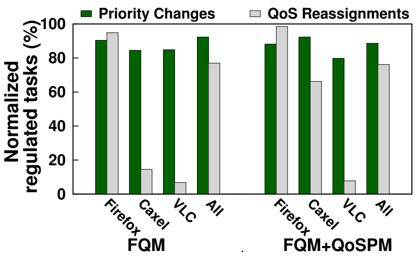
A



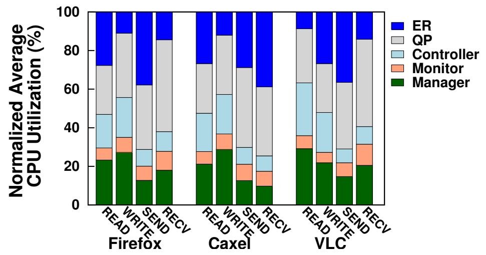
B



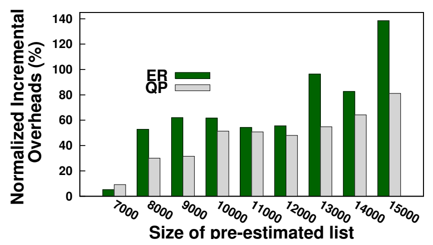
C



D



E



F

Fig. 4. **A** - Average system power consumption (smaller is better); **B** - System power consumption in the first 300 second period in different configurations; **C** - Variation of two major controlled variables: Miss ratio (M) and Utilization ratio (δ) in 300 second period; **D** - Percentage of tasks regulated by FQM; **E** - Average actual CPU cycles to process tasks; **F** - Incremental overhead imposed by ER and QP when its sizes are increased. These incremental percentages are normalized by the baseline scenario where each ER and QP includes 6000 elements.

A. *Power consumption*

Figure 4-A clearly shows that FQM improves the energy efficiency of the BP system. Specifically, our experimental results demonstrate that, in the scenarios of individual or multiple applications (*All Apps*), the system with FQM consumes less power than the others.

When applications are performed individually, QoSPM averagely reduces power consumption by 3% on Firefox and up to 5% on *VLC* and *Caxel*. In contrast, FQM increases these numbers by up to 15% on *Firefox* and *Caxel* whereas 9% on *VLC*. Note that the differences in power saving are mainly caused by the different behaviors of these applications. The I/O transactions of *Firefox* consist of a significant number of I/O operations, which intensively involve with storing data into the cache disk. However, since *VLC* is run from the command line, without caching into disk and without locally displaying, it uses less power. In comparison with *VLC*, *Caxel* transfers a larger amount of data, but uses less energy than *VLC*. Although *Caxel* and *VLC* both use 30 MB system memory to cache data, the I/O transactions of *Caxel*, which mainly write to the disk, create fewer tasks than those of *VLC* that mainly send data back to NIC. Since the NIC is used at the highest capacity, 4.3 W [21], and the disk is operated at the low power mode, 2.3 W, most transactions of *VLC* via NIC cause more power consumption than those of *Caxel*. When FQM is installed, it regulates all tasks on disk and NIC, and hence significantly reduces more power consumption on Caxel than VLC.

When applications are performed simultaneously, FQM helps QoSPM-based system save more power, from 2% to 8%. In detail, due to multiple applications are performed, the number of tasks created and processed is greatly increased. However, these tasks, which belong to different I/O transactions, are not similarly regulated in different systems. A system with FQM not only differentiates tasks based on their sources, but also their targeted devices and I/O operations, while a QoSPM-based system only considers particular metrics of I/O transactions, such as latency and throughput. Therefore, we can see that tasks are regulated more carefully in the system with FQM than the QoSPM-based system. In particular, for those tasks that aggressively occupy CPU, the regulation of FQM reschedules and forces the tasks to execute at lower priorities, thus avoiding the CPU saturation, which causes the most considerable power consumption of the system.

It is clear that FQM on QoSPM-based system is effective at a twofold task regulation: particular transaction metrics specified by QoSPM and multiple application request parameters specified by FQM. Figure 4-A also shows that FQM on QoSPM-based system can save up to 20% energy on Caxel and 13% on VLC. On average, the reduction of consumed power is 10% when applications are simultaneously performed.

Figure 4-B illustrates the variation of system power consumption under different system configurations, where we simultaneously perform all applications and observe the consumed power of system in a period of time. In the vanilla system, the consumed power mainly remains at the upper bound of BIOS PM. This is because of simultaneous I/O transactions, which aggressively increase the CPU utilization [15]. In the QoSPM-based system, we observe that if particular metrics of applications, such as inter-arrival time and throughput, are preconfigured with default values [16], the transactions of these applications are moderately regulated. This regulation impacts upon the system behavior, and hence slightly modifies the consumed power of the system. For example, the QoSPM-based system consumes less power than the vanilla system at the period of 20±10 seconds, which corresponds to the initialization of the first transactions of applications, and the period of 101±10 seconds, which corresponds to the complete of those first transactions.

Without QoSPM, FQM can effectively stabilize the power consumption of the system. This is because FQM reschedules new tasks by changing their priorities, without reassigning their QoS parameters. This reschedule noticeably lowers 5% of the power consumption compared to QoSPM-based system.

With QoSPM, our results show that FQM can also successfully moderate the power consumption of the QoSPM-based system. At the period of 50±20 seconds, we observe that FQM helps QoSPM-based system keep the consumed power stable at a low level. This is due to the twofold task regulation by both QoSPM and FQM. Therefore, when FQM is employed on QoSPM-based system, the power consumption is less varied than that in prior configurations. The power consumption is effectively maintained at the level of 34 W, which is 3% less than that in a vanilla system with FQM installed.

B. *Quality of Services*

For quality of services, our experiments focus on QoS policies applied on tasks. The experimental results show that FQM works well to guarantee application QoS requirements in the vanilla and QoS-based systems. In particular, we focus on the variation of control variables, which include the miss ratio and the utilization ratio, when applications are simultaneously performed. Note that the applications initiate multiple I/O transactions, while these variables are computed based on each transaction. We average the values of these ratios computed based on current active transactions. By observing the variation of the average value upon the increase of tasks, we can validate that whether FQM can steadily guarantee application QoS requirements. Figure 4-C illustrates the variation of these control variables in different systems.

The miss ratio in the vanilla system varies when the number of tasks increases. Even though the maximum value of the miss ratio is 33%, FQM on average maintains this ratio at 18%. In the QoSPM-based system, FQM slightly lowers the maximum value of the miss ratio to 29% and averagely maintains the ratio at 23%. Whereas the average miss ratio is slightly increased in the QoSPM-based system compared to that in the vanilla system, we observe that FQM helps the QoSPM-based system well stabilize the miss ratio. In the vanilla and QoSPM-based systems, the maximum value of the miss ratio corresponds to 250,000 tasks, which is due to transaction modifications and initializations made by users. For example, an user initiates another transaction in *Caxel* while modifying the number of connections of previous transactions. We also observe that FQM does not aggressively reduce the miss ratio when the number of tasks is less than a threshold, e.g. 48,000 tasks. It is challenging to determine the appropriate threshold, because it highly depends on the system environment, which could be affected by the concurrency control in filesystems and congestion control in networks [9].

The variation of the utilization ratio is clearly related to that of the miss ratio. Specifically, failed tasks cause an increase of the miss ratio and a requirement of QoS parameter reassignments for new tasks. The utilization ratio will increase only if these new tasks belong to previous transactions. Otherwise, the tasks of a new transaction will likely cause a decrease of the utilization ratio since it is not properly regulated. When FQM is installed in the vanilla system, the results demonstrate that the utilization ratio varies from 75% to 86% but averagely maintains at 81%. In the QoSPM-based system, FQM increases the utilization ratio by 2%. Such a minor improvement is because QoSPM only monitors 14 transactions, which is slightly insignificant compared to the number of processed tasks.

Regarding the task regulation based on QoS parameter reassignments and task priority readjustments, we quantify the regulated tasks. We examine two cases, FQM in the vanilla system and FQM in the QoSPM-based system. As shown in Figure 4-D, the number of regulated tasks is categorized by applications. In general, the priorities of most tasks are readjusted, but not all QoS parameters of these tasks are reassigned. In particular, for the priority readjustment, 90% of tasks in *Firefox* and 84% of those in both *Caxel* and *VLC* are rescheduled. This rescheduling helps the system avoid CPU saturation, and thereby reduces power consumption.

For *Caxel*, the results show that FQM in the QoSPM-based system reassigns QoS parameters up to 66% of tasks, but only 14% of those in the vanilla system. This is because the inter-arrival time, a particular transaction metric of QoSPM, is set to 0 ms as default, while the transaction requires to open up to 50 connections when it starts. Due to this intensive resource requirement of the transaction, FQM must regulate the tasks, which belong to this transaction, to guarantee QoS without raising power consumption.

For *Firefox* and *VLC*, the number of the QoS parameter reassignments is similar in both the vanilla and QoSPM-based systems when FQM is installed. This is mainly because the I/O transactions of these applications are highly user interactive, resulting in that most tasks of these transactions need to be regulated by FQM. These results are similar to the cases when applications are simultaneously performed.

C. *System performance*

Regarding the system performance, we inspect the overhead induced by FQM. The overhead is known as excess power cycles incurred on system components, such as a processor and devices, resulting in extra power consumption on the system [22]. Since the majority of excess power consumption is attributed to computation and storage operations, a careful examination of this overhead is required. Whereas FQM does not require to store itself in the filesystem, we need to consider the overhead imposed by its computation operations. Note that FQM mainly operates at the Linux kernel level and cannot entirely manipulate I/O transactions. For example, the data size specified in a task is determined by an application and cannot be modified by FQM. Multiple I/O transactions of such an application can create multiple tasks with different data sizes. If FQM processes these tasks to store data into the filesystems, `write` operations of these tasks could potentially increase power consumption by polluting data caches in the processor [22].

As external components, ER and QP could modify their sizes. The sizes of these components depend on their corresponding pre-estimated fine-grained values, such as S_i in ER and `To` in QP. While FQM does not require to directly interact with the system storage, the internal components, such as Manager and Controller, involve with significant computations to monitor the control variables and to refer policies in QP and references in ER.

We measure the actual CPU consumption on each FQM component by individually performing applications. As shown in Figure 4-E, the majority of computations, known as the overhead imposed by FQM, occurs at Manager, QP, and ER. Furthermore, after examining internal components, we observe that the induced overhead varies in different applications. For example, Manager takes 18% of the overhead in *Caxel* and 21% in *VLC*. This is mainly due to the fact that the number of tasks being regulated in *VLC* is larger than that in *Caxel*.

Considering the CPU cost in referring QoS policies and energy references, QP and ER take from 25% to 47% of the total FQM computation. This is due to (1) the size of QP and ER and (2) the complexity of the binary search employed on these external components.

With respect to the size of the pre-estimated list ER and QP, Figure 4-F shows the overhead induced by these components when their sizes are changed. In detail, this figure shows an incremental percentage of the overhead in each component compared to a baseline. Here, the overhead imposed by QP and ER when each component consists of 6,000 elements is defined as the baseline. As default, the value of 6,000 elements is used to conduct experiments on the system with FQM. To increase the number of elements on the components, we lower the fine-grained values of S_i and `To`. We observe that the induced overhead by these components gradually increases with the increase of the number of elements. On average, compared to the baseline, the overheads induced by QP and ER are increased by up to 45% and 26%, respectively, when the sizes of the components increase every 1,000 elements. Regarding the difference between the overheads of QP and ER in each configuration, upon the increase of every 1,000 elements of each component, the difference varies from 8% to 25%.

VI. Conclusion

We propose FQM, a feedback-driven model, to improve energy efficiency of a BP system with the consideration of its QoS requirements. The component-based design allows FQM to be implemented as kernel modules and user applications in Linux systems. We implemented a prototype of FQM and conducted a series of experiments by running real applications. Our experimental results show that FQM can effectively regulate I/O transactions and aggressively exploit CPU cycles, and hence, reducing the energy consumption of the system without degrading the performance of applications. Moreover, our employed optimizations on FQM, such as binary search and a circular read/write buffer, successfully minimize its overhead and save the system's energy consumption by up to 20%.

The work of FQM can be extended in three directions. First, we can adapt FQM in different architectures of BP systems, such as hyper-threading or multi-core processors, to increase the accuracy of the CPU utilization estimation. Second, we can employ FQM on the firmware of devices to reduce the overhead of kernel operations caused by feedbacks. Finally, more investigation is needed to understand the feasibility of FQM in virtual environments. This extension will exploit the virtualization-based system to expand FQM at the hypervisor level and aims to balance power consumption and QoS of running virtual machines.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was partially supported by ONR grant N00014-09-1-0746.

References

[1] "Calculating the battery runtime," www.batteryuniversity.com, [December 2009].

[2] W. Yuan and K. Nahrstedt, "Energy-efficient soft real-time cpu scheduling for mobile multimedia systems," in *ACM SOSP '03*, New York, NY, USA, 2003.

[3] X. Wang and Y. Wang, "Co-con: Coordinated control of power and application performance for virtualized server clusters," in *17th IEEE IWQoS*, Charleston, SC, USA, 2009.

[4] A. Gillen, "The opportunity for linux in a new economy," April 2009.

[5] "Saving power on intel systems with linux," www.lesswatts.org.

[6] "Linux kernel 2.6.23," www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.23 [December 2009].

[7] Y.-H. Lu, L. Benini, and G. De Micheli, "Power-aware operating systems for interactive systems," *IEEE Trans. Very Large Scale Integr. Syst.*, 2002.

[8] Y.-H. Lu, L. Benini, and G. D. Micheli, "Requester-aware power reduction," in *IEEE ISSS '00*, Washington, DC, USA, 2000.

[9] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms*," *Real-Time Syst.*, 2002.

[10] C. Lu, X. Wang, and C. Gill, "Feedback control real-time scheduling in orb middleware," in *IEEE RTAS '03*, Washington, DC, USA, 2003.

[11] C. Lu, X. Wang, and X. Koutsoukos, "Feedback utilization control in distributed real-time systems with end-to-end tasks," *IEEE Trans. Parallel Distrib. Syst.*, 2005.

[12] R. J. Minerick, V. W. Freeh, and P. M. Kogge, "Dynamic power management using feedback," in *In Workshop on Compilers and Operating Systems for Low Power*, 2002.

[13] N. E. Pettis and Yung-HsiangLu, "A homogeneous architecture for power policy integration in operating systems," *IEEE Transactions on Computers*, 2009.

[14] "Onnow pow. mgmt. architecture for applications," www.microsoft.com/whdc/archive/OnNowApp.mspx, [December 2009].

[15] Hewlett-Packard, Intel, Microsoft, Phoenix-Technologies, and Toshiba, "Acpi - advanced configuration and power interface specification," 2009.

[16] M. Gross, "Power management - quality of service," in *ELC 2008*, Mountain View, CA, USA, April 2008.

[17] "Autokey: Text replacement tool for linux," http://autokey.sourceforge.net, [December 2009].

[18] "Alexa 500 sites," www.alexa.com/topsites [December 2009].

[19] "Axel for linux," http://axel.alioth.debian.org [October 2009].

[20] "Vlc media player," www.videolan.org/vlc [November 2009].

[21] Intel, "82563eb/82564eb gigabit platform lan connect," Tech report, November 2007.

[22] I. Crk and C. Gniady, "Context-aware mechanisms for reducing interactive delays of energy management in disks," in *USENIX ATC'08*, Boston, MA, USA, 2008.