Triage: Balancing Energy and Quality of Service in a Microserver

Nilanjan Banerjee Jacob Sorber Mark D. Corner Sami Rollins † Deepak Ganesan

Department of Computer Science
University of Massachusetts Amherst
{nilanb, sorber, mcorner, dganesan}@cs.umass.edu

†Department of Computer Science
University of San Francisco
srollins@cs.usfca.edu

ABSTRACT

The ease of deployment of battery-powered and mobile systems is pushing the network edge far from powered infrastructures. A primary challenge in building untethered systems is offering powerful aggregation points and gateways between heterogeneous end-points—a role traditionally played by powered servers. *Microservers* are battery-powered in-network nodes that play a number of roles: processing data from clients, aggregating data, providing responses to queries, and acting as a network gateway. Providing QoS guarantees for these services can be extremely energy intensive. Since increased energy consumption translates to a shorter lifetime, there is a need for a new way to provide these QoS guarantees at minimal energy consumption.

This paper presents *Triage*, a tiered hardware and software architecture for microservers. Triage extends the lifetime of a microserver by combining two independent, but connected platforms: a high-power platform that provides the capability to execute complex tasks and a low-power platform that provides high responsiveness at low energy cost. The low-power platform acts similar to a medical triage unit, examining requests to find critical ones, and scheduling tasks to optimize the use of the high-power platform. The scheduling decision is based on evaluating each task's resource requirements using hardware-assisted profiling of execution time and energy usage. Using three microserver services, storage, network forwarding, and query processing, we show that Triage provides more than 300% increase in microserver lifetime over existing systems while providing probabilistic quality of service guarantees.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems, interactive systems, real-time and embedded systems*; D.4.8 [**Operating Systems**]: Performance—*measurements*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*

General Terms

Management, Measurement, Performance

Keywords

Power management, microservers, pervasive computing, sensor networks, low-power computing, quality of service, embedded devices.

MobiSys'07, June 11–13, 2007, San Juan, Puerto Rico, USA.

1. INTRODUCTION

The deployment of battery-powered and mobile systems is pushing the network edge far from powered infrastructures. Untethered, mobile, and multi-hop networks support a wide range of applications from wildlife habitat monitoring [19] and security surveillance [15], to space exploration and disaster management [5]. Such applications require the development of highly efficient, long-lived, and low-cost mobile and wireless systems.

One method for balancing efficiency, cost, and lifetime is to deploy distributed heterogeneous, hierarchical systems, that combine resource-constrained nodes, such as Motes [23] with resource-rich *microservers* [10]. Microservers form an intermediate battery-powered tier between tethered basestations or access points, and resource constrained nodes. Microservers provide services including complex data processing [16] and high-capacity storage to augment storage-limited nodes [2]. They respond to user queries in a low-latency manner [16], provide greater range and coverage [15], and act as a gateway between short range and longer-range radio networks [10].

Designing long-lived microservers poses a unique challenge: *how can a microserver provide both performance guarantees and a long lifetime*? These goals are in direct conflict with one another: supporting performance guarantees requires examining each request immediately, as it arrives; however, providing such vigilance is energy intensive. To see why, consider the following: normally the microserver remains in a low-power sleep mode. When a request arrives, it transitions to a higher-power state, determines if it must process the request and returns to a low-power state. Unfortunately, in current microserver designs this duty-cycling process is extremely energy intensive—an effect we validate in the second section of this paper. We propose to resolve this tension through a new architecture, *Triage*, that provides the responsiveness of an always-on server, together with an order-of-magnitude greater lifetime than existing microservers. Triage is predicated on the fundamental prop-

erty that rather than choosing a single hardware platform, the needs of untethered embedded applications are best met by combining platforms with complementary hardware characteristics.

Research Contributions: Triage provides a general mechanism for building highly energy-efficient and QoS-aware microservers suitable for many untethered applications, including sensor applications, mobile systems, and pervasive computing. Our design and implementation offers three novel contributions.

First, the core of Triage is a new system architecture designed for a combination of platforms with complementary characteristics. The architecture is designed for multi-tiered platforms and uses intelligent schemes to distribute tasks across tiers to save energy. The Triage system targets common server functions such as storage, forwarding and query processing.

Our second major contribution is a set of lightweight, online profiling and scheduling mechanisms. A key challenge that Triage addresses is how to place the complex profiling and scheduling logic at the resource constrained platform in order to maximize energy-efficiency without sacrificing QoS. We have built a highly optimized in-situ profiling service, and two intelligent scheduling algorithms: (a) a soft-realtime scheduler that uses an As-Late-As-Possible (ALAP) scheduling policy to provide deadline guarantees to tasks at minimal energy cost, and (b) a lifetime scheduler that uses a token-bucket energy rate control algorithm to achieve a target microserver lifetime while maximizing the number of tasks that can meet their QoS requirements. This scheduler is adapted from previous work on scheduling network transfers in mobile networks [3].

Third, we have built a prototype of Triage on a platform combining the Intel/Crossbow Stargate with the TelosB Mote, augmented with a custom fabricated interface board for in-situ profiling. On this platform, we show the results of an extensive set of experiments using different workloads. We show that Triage provides a 300% increase in microserver lifetime over existing systems. It provides probabilistic Quality of Service guarantees and in addition meets all lifetime goals.

2. MICROSERVER MEASUREMENTS

Microservers typically wait, in a low power state, for an external request, transition to a high-power state, then process the request. For the microserver to be useful, it must have sufficient computational, networking, and storage resources to complete the request. Unfortunately, platforms that have sufficient resources to act as a microserver incur large idling energy costs, even in low-power modes, and can require significant energy to transition from a low-power to a high-power state.

Many systems provide a trade-off between the energy needed for waiting in a low-power state and the cost to transition to a mode that can process the request. As an example, consider the PDA-class Stargate platform [6]. Though it does consume little energy when in a suspended state, it requires several seconds and a large amount of energy to resume operation. This cost is due to a variety of factors including operating system complexity, the use of large RAM banks, and processor startup. In contrast, the Stargate can save energy while waiting using shallower power saving modes, such as CPU DVFS [8], or wireless PSM [1]—these modes require much less energy to transition to an active state, but consume large amounts of power while waiting for requests.

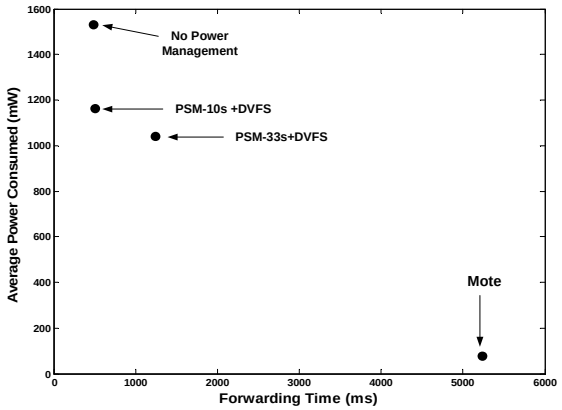

Figure 1: Results of the measurement study

In contrast to the high costs of PDA-class devices, small nodes such as Motes provide very efficient idling and power state transition. This is primarily the result of using a low-power set of components, including a small, simple microcontroller, a low-power radio, small amounts of RAM, and a minimal operating system. However, low-power platforms have insufficient resources to act as a microserver.

We validate these claims through measurements of two possible microserver configurations, a Stargate and a Mote. In each case, the microserver receives a request from a nearby node and responds with a 30kB message. This process repeats for 15 minutes. The Stargate microserver responds using an 802.11 radio, and the Mote uses an 802.15.4 low-power radio. The Stargate can use PSM-10s+DVFS, and PSM-33s+DVFS (the lowest-power PSM mode supported by our card). The 10s, and 33s refers to the wakeup intervals for the WiFi card in PSM mode — i.e., the WiFi card wakes up every 10s, and 33s to scan for incoming packets. The Mote does not use any power management. We show the average power each platform consumes, versus the amount of time required to forward the message in Figure 1. In this case, we do not suspend, or shutdown the Stargate, as the transition times of several seconds would not improve its performance.

The Stargate is a much more capable platform, with a faster radio and processor, yielding very fast forwarding times. However, the high power costs of keeping the platform awake, even with aggressive power-management, gives a very high average power. In contrast, the Mote uses very little power, but requires an order-of-magnitude more time to forward the packet.

For these measurements, we see that neither of the platforms are sufficient to use as an energy-efficient microserver. While the microserver must contain the resources of the Stargate to process low-latency requests, it is highly inefficient when waiting for requests. Our goal is to extend the operating modes of a high-power platform into the regimes offered by platforms such as the Mote, while retaining the resource-richness of the Stargate.

3. TRIAGE ARCHITECTURE

Triage provides Quality of Service (QoS) and energy-efficiency

through a combination of a high-power, resource-rich platform and a low-power, resource-constrained platform. The low-power tier, or *tier-0*, remains always-on ensuring responsiveness at minimal energy cost. The high-power tier, or *tier-1*, remains in a power saving mode until its resources are required for a given service. The low-power platform acts similar to a medical triage unit, examining requests to find the critical ones, and as a scheduler to minimize the number of times the high-power platform is woken. Such a scheduling mechanism requires accurate *in-situ* profiling of the time and energy needs of each task, which is performed at the low-power platform. The scheduler can optimize for different criteria—in this paper we focus on two: minimizing energy consumption while meeting soft-realtime latency constraints, and meeting a lifetime goal while satisfying as many task deadlines as possible.

3.1 Hardware Architecture

Triage employs a *tiered* hardware platform. The tier-0 platform is a very low-power platform, in this case a TelosB Mote [23], and tier-1 is a more capable and higher-power platform, in this case a Stargate [30]. In Triage, the two tiers are tightly coupled and directly communicate over a wired link. This enables the lower tier to trigger the wakeup of the higher tier when necessary.

The TelosB Mote is extremely resource-constrained, but consumes less than one-tenth the power of the Stargate. This platform works well for always-on operation, simple packet processing, and providing low-latency responses. The Stargate platform is significantly more capable but less responsive due to the high latency of sleep to active transitions. We augment this two-tier platform with a custom fabricated interface board that provides necessary voltage conversions for the two platforms, the wakeup interface, as well as a fuel-gauge chip to measure energy-consumption in-situ.

3.2 Software Architecture

Figure 2 illustrates the components of the software architecture. Tier-0 virtualizes resources available on tier-1 using a collection of *surrogates*. Surrogates receive requests from the network and can service them in three ways: by using locally cached information, by performing local execution, or by passing them to tier-1 for execution. The surrogates decide on how to service the request based on information provided by the profiler and the scheduler. A *profiler* measures the energy and processing requirements of tasks and a *scheduler* determines, based on the predicted energy cost of a task, when and where it should be executed to meet QoS requirements. Requests that have been scheduled for execution at tier-1 are written into a *log* and delayed until their scheduled execution time. This log also serves as a *cache* of recent requests.

3.2.1 Surrogates

Surrogates are small software modules that run on tier-0 and provide a service such as storage or forwarding. Though tier-0 can process simple tasks, such as routing updates or time synchronization, most tasks require resources only available at tier-1. When a request arrives at the microserver the surrogate performs the following process: (1) immediately execute requests from information cached at tier-0; (2) if a request cannot be serviced from the cache, ask the scheduler to determine where and when to execute the task; (3) if the scheduler determines that the task should be executed at tier-0, execute it immediately; (4) otherwise write the request into the delayed request log. The delayed request log acts as a priority queue—the task with the smallest deadline lies at the head of the queue.

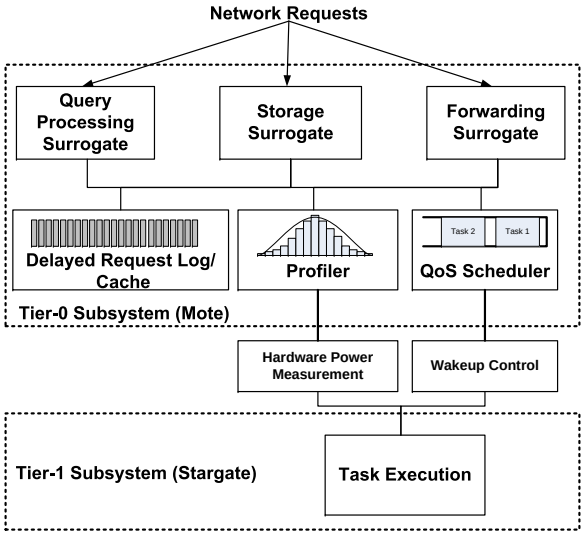

Figure 2: Microserver Software Architecture

Triage also uses the delayed request log as a cache for the surrogates. This functionality is particularly useful in storage applications; a read closely following a write to the same data can be serviced from the log. In order to maximize the amount of cached data, Triage does not erase the tier-0 log when a batch of requests is played at tier-1. Instead, the previously committed log entries and cached results are lazily overwritten by new requests using an LRU eviction policy. Though writing into the log consumes energy, we argue that it is insignificant to the savings from minimising the number of tier-1 wakeups.

To enable applications to compose the functionality of several surrogates, Triage also permits communication between surrogates using primitives provided by the operating system. For instance, a client may query the microserver for information, and request that the results of the query be sent to another node. This requires a combination of a storage surrogate as well as a forwarding surrogate.

3.2.2 Scheduler and Profiler

Triage uses a scheduler, running on tier-0, to provide QoS. The scheduler relies on a profiler to provide information regarding how long each type of task will take to process, and how much energy it will consume. The profiler measures the execution time of each task and builds a model of task execution time. Using this information, the scheduler determines *where* and *when* to execute each request. The scheduler relies on the execution time profiles generated by the profiler. However, if a task exceeds the typical execution time, it is not preempted and still carried on till completion.

The question of where to execute a task can be answered by comparing the amount of energy required to execute it at each platform. This decision may be even simpler if the task requires the resources of tier-1 and cannot be executed

on tier-0. The question of when to execute requests is more complicated. There are two cases when the scheduler must wake tier-1 and dispatch outstanding tasks. The first case occurs when the log becomes full. In this case, the scheduler is automatically invoked by the delayed request log; it wakes tier-1 and dispatches each outstanding task to the appropriate service. The second case requires the scheduler to consider QoS constraints. Each request can optionally contain a soft-realtime deadline, or latency constraint, which indicates the time by which a task should be executed. In order to meet the deadlines with maximum energy efficiency, the scheduler will delay execution of tasks as long as possible to increase the amount of time tier-1 remains in a low-power state. However, if a batch is already being processed, all tasks irrespective of their deadlines are executed on tier-1. This is done to avoid the high transition cost of waking up tier-1. We describe the algorithms used by the scheduler and profiler in Section 5.

4. EXAMPLE SURROGATES

Some of the common, and basic, functions found in servers for sensor networking, mobile networking, and pervasive computing are storage, query processing and forwarding. To this end, we present three example surrogates: a storage system surrogate, a network forwarding surrogate, and a query processing surrogate. As untethered networks proliferate, this library of surrogates will be expanded, enhanced, and further optimized.

4.1 Storage System Surrogate

The storage surrogate enables in-network storage applications. It accepts read, write, and delete requests for the tier-1 storage system. Upon receiving a request from the network, it first determines whether the request is a read request that can be satisfied by a recent write cached in the delayed request log. If so, it immediately provides the result. Otherwise, it asks the scheduler to schedule the task. The scheduler considers any soft real time deadline provided with the task and tells the surrogate when the task has been scheduled. The surrogate then inserts the task into the delayed request log.

4.2 Network Forwarding Surrogate

The network forwarding surrogate enables efficient routing by utilizing both the tier-0 and tier-1 network interfaces. When a packet arrives at the surrogate, it examines the destination address, consults a routing table, and determines over which radios the destination is reachable. It immediately passes this information, along with any latency constraint, to the scheduler which determines which radio should be used to send the packet and when the packet should be sent. If scheduler determines that the tier-0 radio should be used, the packet is sent immediately. Otherwise, the packet is inserted into the delayed request log.

4.3 Query Processing Surrogate

The query processing surrogate provides a simple query interface for data stored on the microserver. Clients may use simple queries, such as *retrieve all images from the last ten seconds*, or more complex queries, such as *retrieve all images that contain 2 or more objects and are from a particular geographic region*. The queries are specified by the following fields (1) the type of query (simple/ complex) (2) the images queried (3) the number of objects desired per image for complex queries (4) latency deadline associated with each query. The query processing surrogate uses the other surrogates to create a complex combination of resources, including processing, routing, and storage. Tier-1 can execute any query since it has access to the powerful radio, the primary storage system, and a powerful processor. However, tier-0 can respond to only simple queries. For example, any query that can be performed using simple comparisons of cached metadata can be performed at the tier-0 system. Therefore, tier-0 maintains an index of results stored from previous queries in its cache/log.

When the surrogate receives a query, it first determines whether the query is a simple query that can be executed over data cached in the delayed request log. We assume that tasks are statically mapped into simple queries, which can be executed at either tier, and complex queries that require the resources of tier-1. If the query can be executed at tier-0, it is executed immediately. Otherwise, the surrogate passes the query and any latency constraint to the scheduler. Once the scheduler has scheduled the query, it is inserted into the delayed request log.

5. PROFILING AND SCHEDULING

At the heart of Triage is a profiling engine that is used to estimate the execution time and energy usage for different tasks, and a scheduling engine that determines how to meet QoS constraints. In this section, we describe the algorithms employed by the profiling engine and the scheduling engine to enable energy-efficient scheduling of tasks.

5.1 Task Profiling Algorithm

Triage employs a profiler to measure the execution time and energy usage for different tasks. In this discussion, we focus on determining the *typical energy usage* and *typical execution time* for each type of task. Such online profiling is necessary to deal with the variability in execution time and energy usage of tasks that involve a combination of processing, communication and storage.

Online profiling involves two steps, task grouping and parameter estimation. The online profiling engine first identifies a task as belonging to a certain group based on the nature of task. This grouping information is assumed to be provided a priori by the system designer. We believe that such a grouping is appropriate since many applications of microservers involve a small and well-specified set of tasks. For instance, in a camera sensor network, a typical set of tasks might be {`Motion Detection`, `Face Recognition`, `Store Image`, `Send Data`}.

For each of these task groups, the profiler uses a separate history of execution times and energy consumption to build corresponding probability distributions. We focus on the estimation of the typical execution time since a similar algorithm can be used for estimating typical energy usage.

Let $f(t_i)$ be the probability distribution of time taken to execute task type i. Further, let $\overline{X}_i$ and σ_i denote the average time and standard deviation for task type i. The profiler uses the Chebyschev's inequality (shown in Equation 1) to determine an interval of time such that the task executes within that interval with probability at least p. Triage consequently takes the upper bound on the interval as the typical execution time for the task. This is a conservative estimate, however for tasks whose execute time distributions are not

known apriori, a conservative strategy is warranted. Moreover, typical execution time refers to the time such that p% of the requests are likely to be satisfied.

$$Time(i,p) = [\overline{X}_i - \frac{\sigma_i}{\sqrt{1-p}}, \overline{X}_i + \frac{\sigma_i}{\sqrt{1-p}}] \quad (1)$$

The parameter p can be tuned depending on the guarantee required by a user. For instance, in the camera network used for surveillance, a `Face Recognition` task might require a tight guarantee as the person might move out of the field of view of the camera sensors. In this case, p can be set to a high value, say 0.9. Other tasks such as `Send Data` might be more elastic, and `Store Image` might not have any deadline at all.

Using information collected after the task executes, the profiler builds two kinds of dynamic models: histograms and parametrized models. When prior models of execution time are unavailable, a simple histogram approximates the probability distribution, $f(t_i)$, and tracks bins of execution times and energy consumption for each task group. In contrast, when prior models are available, these can be used to more accurately model task execution time and energy consumption. For instance, the execution time and energy used in a communication task is a linear function of the number of bytes transmitted. In this case, the execution times are fit to a simple linear model to determine the costs of the two radios in Triage. Our prototype uses parametrized models for the storage and network surrogates tuned to the size of the file and the size of the packet respectively. More complicated models can be built for applications such as video coding, compression, and encryption. For the query processing surrogate we use the histogram-based profiling.

5.2 Task Scheduling Algorithm

The scheduler that resides on tier-0 uses the profiling information about tasks to minimize the number of times tier-1 is woken while still satisfying the task deadlines. The Triage scheduler uses different algorithms depending on the optimization criteria. In this work, we discuss two schedulers — the first is optimized to satisfy task deadlines, and the second is optimized to achieve a target lifetime for the microserver. While we limit our discussion to these two schedulers, we note that alternate schedulers that optimize, or balance, other QoS constraints can be plugged into our system.

5.2.1 Scheduling for Deadline Constraints

The deadline scheduler tries to minimize the energy consumed by the microserver, such that the deadlines of the incoming tasks are met. Let the set of tasks which are already batched at tier-0 for delayed execution at tier-1 be denoted by $S = \{T_1, ..., T_k\}$ where task T_i has deadline $D(T_i)$, latest start time $L(T_i)$, and execution time $E(T_i)$. The latest start time is the latest time at which a task can begin executing on tier-1 such that the deadlines of all tasks after and including itself are met, and the execution time is the time it takes to execute the task on tier-1. $L(T_i)$ and $E(T_i)$ are provided by the profiler. Further, we assume that the list S is sorted by deadlines *i.e.* $D(T_i) > D(T_j)$ if $i > j$. Let the wakeup time for tier-1 be W, and the current batch time, B, correspond to the latest time at which tier-1 needs to be woken up such that the deadlines of all tasks in the list S can be satisfied.

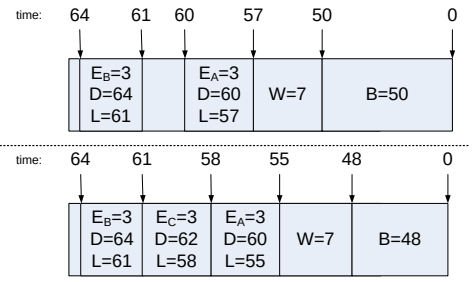

Figure 3: ALAP Example: The figure shows the execution time and deadline for each task, the wake up latency for tier-1, and the resulting batching time. A new task T_C is inserted into the scheduling decreasing the batch time.

The scheduling framework that we propose is based on the well-known As Late As Possible (ALAP) scheduler [29]. When a new task arrives, the scheduler first queries the profiler for the typical execution time for the task at tier-1. Next, the algorithm recomputes the batch time, B, *i.e.* the latest time at which tier-1 can be woken such that all the batched tasks and the new task meet their deadlines. Let the new task be inserted at index l into the sorted list S based on its deadline. The scheduler now needs to ensure that the insertion of the new task does not result in missed deadlines for any of the other tasks in the list. The scheduler only lowers the batch time and never increases it, hence only the tasks that are before T_l in the list need to be checked for deadline violation. Thus, for each task $T_i : l \geq i \geq 1$, the scheduler sets the latest start time such that it does not violate the deadline constraint of T_i or any task with deadline after T_i, *i.e.* $L(T_i) = min(L(T_i), L(T_{i+1}) - E(T_i))$. The new batch time, B, is updated to reflect the latest start time for the first task in the list, *i.e.* $B = L(T_1) - W$. If $B \leq 0$, tier-1 is immediately woken and the batch executed. If $B > 0$, a timer will fire at time B and tier-1 will be woken. The time required to update the schedule is linear in the number of tasks currently in the queue.

We illustrate the deadline scheduling algorithm with a simple example, shown in Figure 3. Let there be two batched tasks, T_A with deadline 60 seconds and execution time 3 seconds, and T_B with deadline 64 seconds and execution time 3 seconds. The latest start times of the two tasks are $L(T_A) = 57$ seconds and $L(T_B) = 61$ seconds respectively, and the batch time, B is 50 seconds, assuming a tier-1 wakeup time, W of 7 seconds. Now, a new task T_C arrives with deadline 62 seconds and execution time 3 seconds. The task is inserted between T_A and T_B. The scheduler checks whether the current batch time satisfies T_C's deadline, notices a violation, and pushes T_A forward in the schedule. Hence, the batch time is set to 48 seconds.

5.2.2 Scheduling for a Lifetime Constraint

While the goal of the deadline scheduler is to miss only a small percentage of deadlines while minimizing energy usage, the objective of the lifetime scheduler is to meet a target lifetime for the microserver while satisfying as many deadlines as possible. The scheduler should also be capable of handling periods of burstiness. To accomplish this we use a token-bucket algorithm for the lifetime scheduler. We use

a lifetime scheduler from our previous work on networking relays [3]. We summarize the algorithm here. Given a target lifetime, L, and battery capacity, E, energy tokens are generated at a constant rate of $\frac{E}{L}$. The total number of accumulated tokens represents the amount of energy that is available for use by the system. The amount of energy used by the system is continually monitored by the energy profiler, and is queried periodically by the scheduler to determine the rate at which energy is depleted by the system. The difference between the accumulated energy tokens and depleted energy tokens at any time represents the surplus of energy that can be used by the system to execute the batched tasks.

The lifetime scheduler builds on the deadline scheduling algorithm that we described in Section 5.2.1. When a new task arrives, the deadline scheduling algorithm is used to queue the task and determine the batch time. When this batch time becomes zero, the lifetime scheduler checks to see whether there are sufficient energy tokens to wakeup tier-1, execute all the tasks in the batch, and shutdown tier-1. If so, tier-1 is woken and the tasks are executed before shutting it down. The energy profiler is queried to determine how much energy was used during this batched processing, and the number of available energy tokens is updated accordingly. If the number of energy tokens is insufficient to execute the batch, the wakeup of the tier-1 platform is delayed until sufficient tokens have accumulated. During this period of waiting for tokens to accumulate, task deadlines could be missed, and tasks could be dropped if the size of the task queue exceeds the storage capacity of the tier-0 platform.

Neither of our scheduling algorithms take into account the availability of DVFS states at tier-1. Evaluating and profiling multiple DVFS states would possibly permit greater efficiency by allowing tier-1 to sleep longer, however this pushes the complexity of the scheduler to $O(k^n)$ for n tasks and k power modes — the scheduling problem can be shown to be NP-Hard. We are currently investigating approximate heuristics that are computationally feasible for the tier-0 node. These heuristics can be implemented on top of the scheduling algorithms described in this section to determine when tier-1 should be woken up.

5.2.3 *Idle State Management*

One remaining issue is the state in which the scheduler leaves tier-1 while it batches new requests at tier-0—there are two options, suspension and shutdown. Suspension requires more idle power than shutting down the platform, but the transition cost is lower. For example, the Stargate platform with a CF 802.11 card draws 60.46 mW in suspend mode and costs 3.67 J and 32 J of energy to wake up from suspend and shutdown respectively.

The Triage scheduler determines the appropriate idle state for tier-1 based on the expected time between wakeups of tier-1. The expected time between wakeups are calculated over the last k wakeups, where k is a constant. We estimate the cost of each state based on the expected idle time, and choose the state that minimizes this cost.

While this approach is sufficient for many applications, the choice of idle state does enforce a minimum latency that the microserver can support. For example, if tier-1 is shutdown due to infrequently arriving tasks, the next task that requires tier-1 will have to wait at least as much time as tier-1 requires to wake from shutdown. In the case of the Stargate, this minimum latency is 15.6 seconds. In order to deal with this problem, more sophisticated profiling techniques could be used to anticipate the latency requirements of upcoming tasks and choose the proper idle state accordingly.

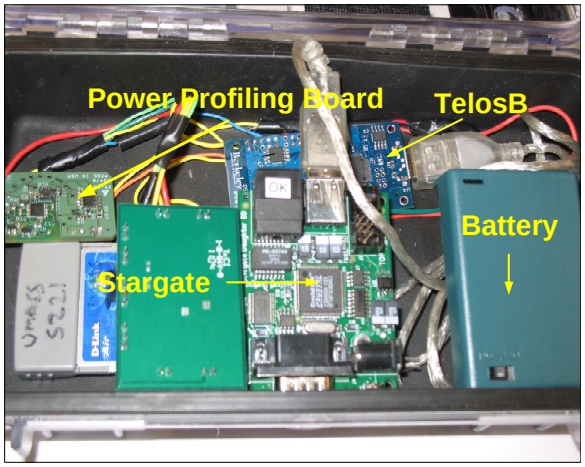

Figure 4: Prototype Triage System

6. IMPLEMENTATION

In order to evaluate our approach we have implemented a working prototype of Triage, shown in Figure 4.

6.1 Prototype Hardware

We constructed a prototype using a Crossbow Stargate (tier-1) [30] and a TelosB Mote (tier-0) [23]. The Stargate contains a 32-bit, 400MHz PXA255 XScale processor, 64 MB of RAM, 32 MB of internal flash, and a WiFi interface. The TelosB Mote contains an 8-bit, 8 MHz microcontroller, 10kB of RAM, 1 MB of external flash, and an 802.15.4 radio. These hardware platforms were chosen because they handle the range of workloads that we have targeted, are separated in power consumption by more than an order-of-magnitude (300mW-3000mW and 20mW-100mW), are easily programmable, and are well supported. The Stargate platform runs Linux, making available a broad range of software tools and services. We used the power-gating technique with a relay to switch off power to the CF card when the Stargate is suspended [22]. This technique brought down the suspension power of the Stargate to a modest 61 mW.

We used a power supply board designed for attaching both boards to a single battery [28]. This board provides an additional hardware element necessary to Triage, a Maxim DS2770 fuel gauge chip. The fuel gauge chip gives accurate readings of the energy left in the battery, and the amount of energy used by the system during task execution. As described, the scheduler uses this profiling information to control the platform energy policy.

One limitation of our current implementation is the transfer speed from the TelosB Mote to the Stargate. The two devices communicate over a USB line which is limited to 230 kbps. However, data needs to be read from the flash and then transfered over the USB which increases the total time required for the transfer. As a result, transferring 1024 kB of batched work along with protocol overhead takes more

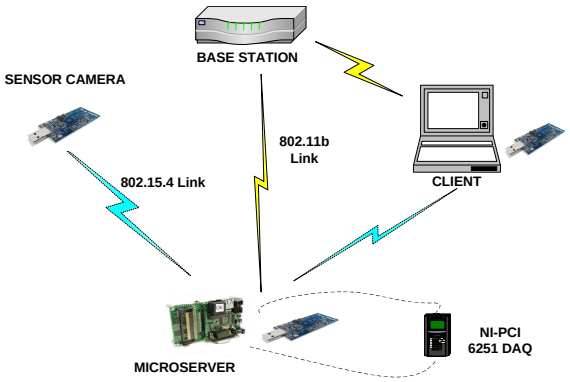

Figure 5: Experimental Setup

TelosB Max. Power Consumption	100 mW
Stargate Bootup Energy	32 J
Stargate Bootup Time	15.6 s
Stargate Resume Energy	3.7 J
Stargate Resume Time	2.6 s
Stargate Suspension Power	60.5 mW
Stargate Idle Power (WiFi in PSM-idle mode)	912.3 mW

Figure 6: Platform Measurements

than 150 seconds and wastes a great deal of energy while blocked on serial I/O. We are currently investigating more efficient means of communication. Even with this limitation the current prototype shows extremely high gains in energy efficiency.

6.2 Surrogates and Log

As part of this prototype, we implemented three surrogates: storage, network forwarding, and query processing. The delayed request log is managed on the TelosB's flash storage using a custom designed file system. We implemented these components as TinyOS modules written in nesC [9]. The forwarding, storage, and query processing surrogates comprise 535, 480, and 540 lines of nesC code respectively, and the custom file system consists of roughly 1600 lines of code. The profiler and scheduler consist of 1360 lines of code. We also implemented an execution engine that runs on the Stargate and executes tasks when they are received from the Mote [1].

7. EVALUATION

We evaluate the performance of Triage through a extensive set of experiments. First, we provide micro benchmarks that validate our use of a two-tier platform to achieve a combination of capability, energy-efficiency and responsiveness. Second, we evaluate the deadline and lifetime schedulers, and demonstrate their effectiveness in achieving their goals. Third, we focus on the performance of the profiler, and its accuracy when used with the forwarding surrogate. Fourth, we demonstrate how the Triage system is able to adapt itself to different QoS constraints at minimal energy consumption. Finally, we show the energy consumption of different independent components of the system and identify potential bottlenecks in the system.

[1]The source code for Triage is available at `http://prisms.cs.umass.edu/hpm/triage.html`.

We use a camera sensor network application in our evaluation. The application involves in-network processing, storage and forwarding and forms an adequate testbed for Triage. The experimental setup is shown in Figure 5. Motes emulate cameras in the network and feed images of variable sizes to the microserver. The other nodes in the network are client Motes attached to a device equipped with a 802.11b interface. Therefore, results of queries can be routed back to a client using a 802.15.4 or a 802.11b link. All power measurements were taken using a NI-PCI 6251 DAQ with a SC-2345 signal conditioning unit.

In our evaluation, we compare Triage with three other systems.

1. PSM-DVFS : This is a single-tiered dual radio system which uses WiFi PSM and DVFS (dynamic voltage frequency scaling) to save power on the Stargate. The system runs a DVFS algorithm which uses previously measured data to identify the lowest DVFS state on the Stargate where the given deadline can be satisfied. The wakeup interval in PSM mode for the 802.11 card is set to the maximum value supported by the card (33 secs). This provides an accurate comparison with a system which does not use the hardware architecture of Triage.

2. WoW* : The system is similar to Wake-on-Wireless [26]. The published Wake-on-Wireless system wakes up when it receives a network packet. WoW*, however, wakes up tier-1 whenever a task arrives at tier-0. Tasks are always executed on tier-1. WoW* is a system which uses the hardware architecture of Triage but does not use its software architecture.

3. Triage-Batch : The Triage-Batch system uses the same hardware as Triage. However, it does not use any online profiling, scheduling or caching. It batches a task as long as its deadline permits and then wakes up tier-1 to execute the task.

7.1 Static Energy Costs

In order to provide better intuition into the behavior of our prototype, we measure the static energy costs that directly impact Triage's performance. These values, shown in Figure 6, are the basis for the energy savings achieved by Triage. We observe that the idle cost of the Stargate platform with WiFi card in PSM mode is 6 times the sum of the suspend power of the Stargate and the maximum power consumed by the TelosB. Therefore, offloading tasks to tier-0 while keeping tier-1 asleep can lead to substantial energy savings. However, the Stargate's transition from suspend to active costs as much energy as 36 seconds of computation on tier-0. Transitioning from shutdown to active is equivalent to 319 seconds of active tier-0 computation. Therefore, replacing expensive state transitions at tier-1 with low-power, tier-0 computation as long as possible can lead to minimal energy costs for the system.

7.2 Soft-Realtime Scheduling

In order to evaluate the deadline scheduling algorithm, we observe how a Triage microserver performs in the presence of different latency constraints. In this experiment the microserver answers queries of client Motes for objects in images. Such a query is common in surveillance applications where a user might want to detect movements at a

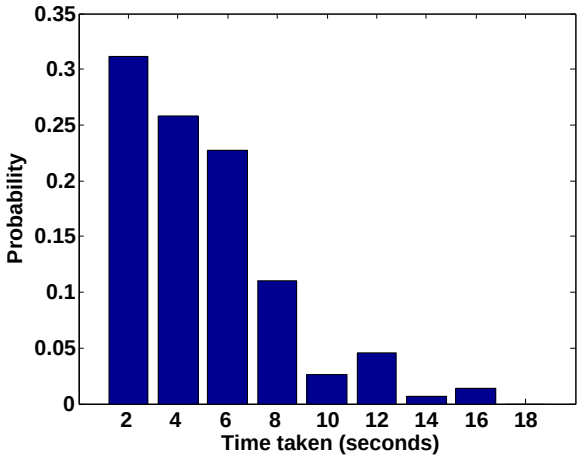

Figure 7: Histogram of the time taken to execute image classifier on Stargate

scene or extract important features of a scene. The objects in the image are computed using the `kmeans` classifier at the microserver and the classified image is sent back to the client.

The histogram of the amount of time taken to classify random 100-by-100 images using the `kmeans` algorithm is shown in Figure 7. The figure illustrates the large variance in the amount of time taken to classify an image. Moreover, it is clear that the time taken to classify an image does not follow a known probability distribution. Hence, accurate time profiling for the application is crucial to the success of the scheduler in meeting the deadline constraint at minimal energy consumption. The profiler uses a threshold of $p = 0.9$ in Equation 1 to determine the typical execution time of each task, i.e., at least 90% of the time the microserver will meet its deadline for tasks.

We evaluate the scheduler on two task arrival distributions—(i). when tasks arrive at a constant rate of one per 30 seconds (ii). when tasks arrive in bursts of three queries—the inter arrival time between bursts follow a Poisson distribution with mean 30 seconds. While the first scenario exhibits a constant load on the system which is easy to learn, the second task arrival distribution tests loads which are unpredictable. Therefore, the system has to accurately predict and adapt to the variable load patterns to perform well.

We varied the latency constraint on the task in the experiment and we measure the power consumed by the microserver and the percentage of tasks completed within the deadline. Each point on the x-axis represents experiments where the latency constraint is chosen uniformly at random within the interval $[x-30, x+30]$ seconds. 100-by-100 images are sent to the microserver at a rate of one per minute. The results are compared with the Triage-Batch system. This is because the Triage-Batch system keeps the tier-1 system in the low-power mode for a longer period of time.

The results of the experiment are shown in Figure 8 and Figure 9. The first thing to note in Figure 9 is that for each workload the Triage deadline scheduler is always able to meet at least 90% of the latency constraints. Triage is able to adapt to bursty unpredictable task arrival distributions. This demonstrates both the accuracy of the non-parametric histogram profiler of Triage, which is able to precisely determine the computational needs required for these queries, and the accuracy of the deadline scheduler, which correctly schedules the wakeup of tier-1 to meet the desired latency constraints. Without profiling the execution time of tasks or the deadline scheduler, the Triage-Batch scheduler is unable to determine when to wake tier-1 and regularly misses deadlines, especially when latency constraints are small. This is because scheduling errors are more likely to occur at the beginning of each batch of tasks. Allowing longer latencies results in larger batches of tasks and more tasks that are processed ahead of their deadlines. However, it is worth noting that the Triage system only consumes slightly more power than the Triage-Batch system.

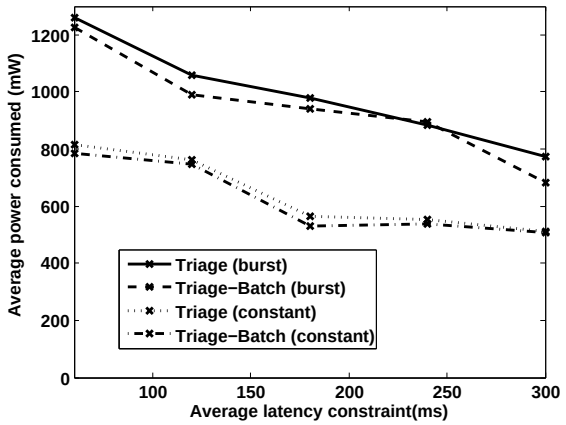

Figure 8: Average Power Consumption for the deadline scheduler. The deadline scheduler consumes slightly more power in order to meet more deadlines.

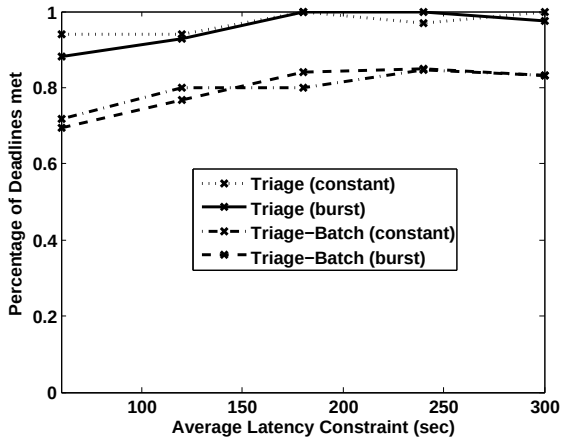

Figure 9: Percentage of Queries executed within deadline. Triage meets more than 95% of all deadlines while Triage-Batch is able to meet only 70% of all deadlines

The second observation from the experiment is that the Triage system satisfies all constraints irrespective of the task arrival distribution. However, both systems consume more power for the bursty task arrival distribution. This is because, even though the mean inter-arrival time between bursts is the same as the interval between arrival of constant rate tasks, the number of tasks executed for bursty task arrival is more than the constant task arrival distribution.

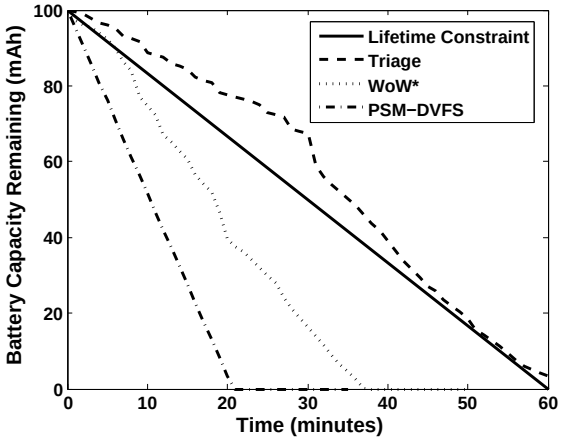

Figure 10: Lifetime Scheduler. This figure shows Triage's use of the lifetime scheduler for a goal of 60 minutes. For the first 30 minutes the load is light and the microserver accumulates an energy surplus. For the last thirty minutes it uses the surplus at the detriment of meeting deadlines.

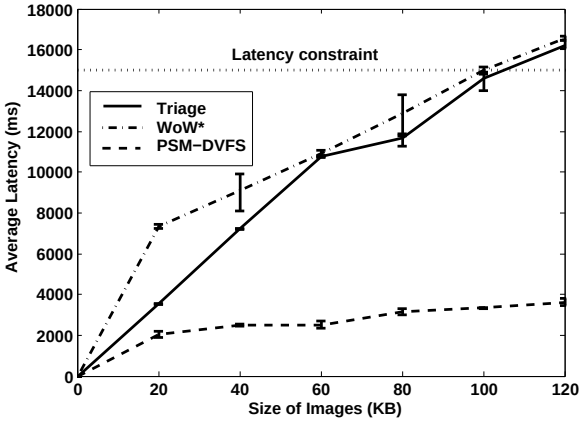

Figure 11: Time Profile Accuracy. This shows the forwarding surrogate choosing between two radios based on a latency constraint. For 80KB of data is switches from the 802.15.4 radio to the 802.11 radio to meet the constraint.

7.3 Lifetime Scheduling

To show that the lifetime scheduler is able to meet a lifetime goal, we subject the microserver to a similar experiment as before using Query Processing. However, we use simple queries for an images stored in the Stargate storage. In order to expedite this experiment, we use a small battery capacity of 100 mAhr—enough energy for the microserver to operate at maximum load for 9 minutes. We set the lifetime goal of the server to be 60 minutes. For the first 30 minutes, all queries arrive at a constant rate of one per 30 seconds with a latency constraint uniformly distributed over the interval $[150, 210]$ seconds. For the remaining time, the server sees a more intense load with queries arriving with latency constraints uniformly distributed over the interval $[5, 15]$ seconds. Figure 10 shows the results of this experiment. The slope of the straight line demonstrates the lifetime goal divided by the energy capacity of the server—this is the overall average power goal.

Recall that when using the lifetime scheduler, Triage prioritizes lifetime over latency constraints. It attempts to meet the latency constraint whenever it can, and the token-bucket algorithm allows for bursts of short, energy intensive workloads. For this algorithm to operate correctly, Triage must accurately profile the energy use of tasks and track the overall energy consumption of the microserver to account for profiling errors. For the first 30 minutes, the server consumes less energy than is required to meet its lifetime goal. During this time the server's workload is insufficient to drain the bucket, so Triage is free to schedule all tasks and therefore meets its deadlines. After operating for 10 minutes under a more intense workload, Triage continues to meet its deadlines, but it begins to consume the surplus energy that has accrued. At 45 minutes, Triage runs out of surplus energy and begins to sacrifice latency constraints for conserving energy. Note that Triage meets the lifetime goal with an excess energy of about 3mAh. The WoW* and PSM-DVFS systems are unable to meet the lifetime constraint. Their batteries die out at 38th and 21st minutes respectively. Therefore, these systems are left without any battery for 37% and 65% of the time.

7.4 Task Profiling

The profiling function of the microserver is essential in providing soft-realtime guarantees. We use simple parameterized linear models for the forwarding surrogate and general histogram models for the more complex tasks like image processing where good models are not known apriori. The model parameters are learned by Triage over time. The efficacy of the histogram model was shown in Section 7.2. We present the efficacy of the parameterized model here. We use an experimental setup where variable sized images are sent from the camera Motes to the microserver at a fixed rate of one image per minute to be routed to some destination Mote. The destination could correspond to a central processing server or another node. Since the linear model for the amount of time taken to route data for the two radios is a function of the amount of data that Triage wants to send over the radio, we vary the size of the images sent during the experiment—this corresponds to images of different resolutions required by a client.

Each forwarding request has a fixed latency constraint of 15 seconds. We show the results of the experiments in Figure 11 and Figure 12. We compare our results with the PSM-DVFS and WoW* systems. Comparing these systems we see several effects. First, Triage is able to profile the time required for forwarding tasks correctly and send them by their required latency constraint for data sizes less than 100KB. Triage uses the 802.15.4 radio at the lower data rates, and above 80KB of data it must wake the tier-1 system to make the latency constraint. Second, as Triage can use the TelosB-node alone to transfer data it can achieve 200% increase in lifetime over WoW* system and 500% increase in lifetime over PSM-DVFS system. The savings clearly demonstrate the benefit of using a tiered architecture. The architecture provides the microserver with the flexibility of using resources on either tier intelligently—leading to substantial energy savings. Third, both WoW* and Triage are unable to meet the latency constraint for 120KB images.

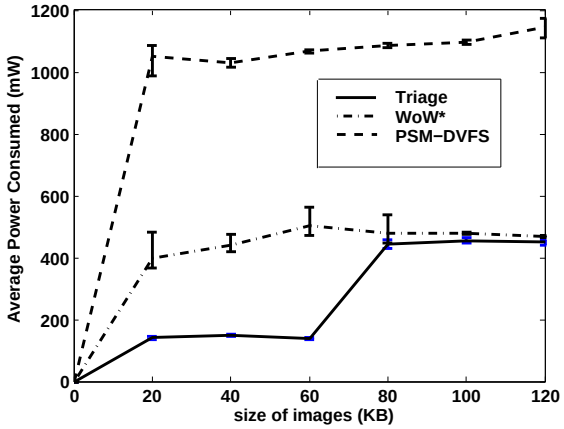

Figure 12: Time Profile Power Consumption. This shows the forwarding surrogate power consumption based on the amount of data it sends. At 80KB of data it must switch to the 802.11 radio to meet the latency constraint, thus using more power. The WoW* and PSM-DVFS solutions use more power as they only use the 802.11 radio to send data.

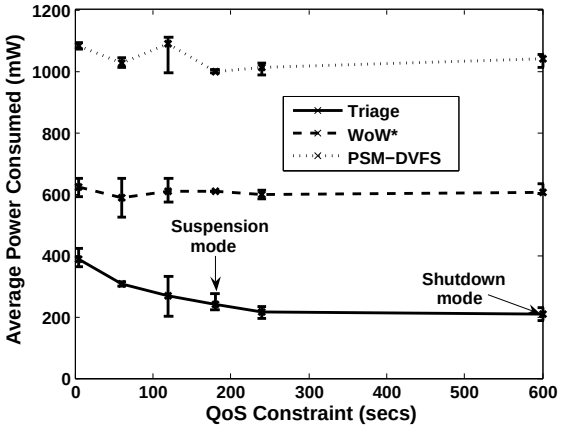

Figure 13: Scaling to QoS constraints. Triage finds the minimal energy cost at which a QoS constraint can be satisfied. The WoW* system and PSM-DVFS system consume 3x and 5x more power than Triage respectively.

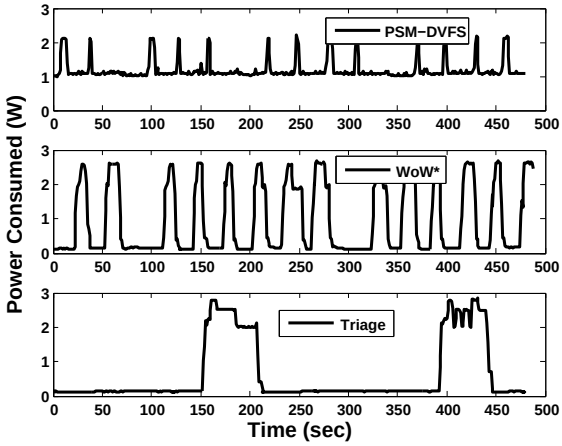

Figure 14: Power Traces for the Triage, WoW* and PSM-DVFS systems.

This is due to the USB data transfer bottleneck for multi-tiered system. The PSM-DVFS system meets all deadlines at a much higher energy cost.

7.5 Scaling to QoS Constraints

The primary goal of Triage is to balance energy consumption of the server with a given QoS constraint. Triage uses a combination of task profiling, scheduling, caching, and idle state management to determine the minimal cost at which a QoS constraint can be satisfied.

We perform the following experiment to validate the above claim. Camera Motes feed 100-by-100 images into the microserver at a rate of one per minute. Client Motes request images from the microserver at a rate of one per 30 seconds. Each query request is for a single image taken T seconds ago, where T is exponentially distributed with a mean 100 minutes. This represents applications where newer data is more valuable to the user than old data. We vary the latency constraint on the task in the experiment. We compare Triage with PSM-DVFS and WoW* systems. The results are shown in Figure 13. We see that Triage consumes 300% less power than WoW* and 600% less power than the PSM-DVFS system. Triage uses a combination of serving requests from cached data, scheduling and delayed execution to amortize the transition cost of the tier-1 platform over a long period of time and hence shows large power savings.

7.6 Component Power Consumption

Finally, we demonstrate the power consumed by independent components of Triage. We perform an experiment where 100-by-100 images are sent to the microserver at a rate of 1 per minute. The client Motes request classified images with a task arrival rate distributed in the interval $[30, 60]$ seconds. This application provides for a combination of storage, data transfer, processing and data forwarding. The power trace collected is shown in Figure 14. The break-down of the average power consumed by different independent components of the system are show in Figure 15.

We find that the PSM-DVFS system suffers from a huge idle cost of keeping the Stargate platform awake with the WiFi card in PSM-idle mode. This problem is solved by the WoW* system by using the hardware architecture of Triage and duty-cycling the Stargate. However, the WoW* system suffers from a huge transition energy cost, since it has to wake up the Stargate on every task arrival. The above problem is solved by Triage using its software architecture and amortizing the transition cost of a long interval of time. However, we find that the USB-transfer cost is a potential bottleneck for both the Triage and WoW* systems and the performance of Triage could be improved if the bottleneck is eliminated using low power DMA (direct memory access) between the Mote and the Stargate.

8. RELATED WORK

The design and implementation of Triage draws from several related research areas.

8.1 Microservers and Clustering

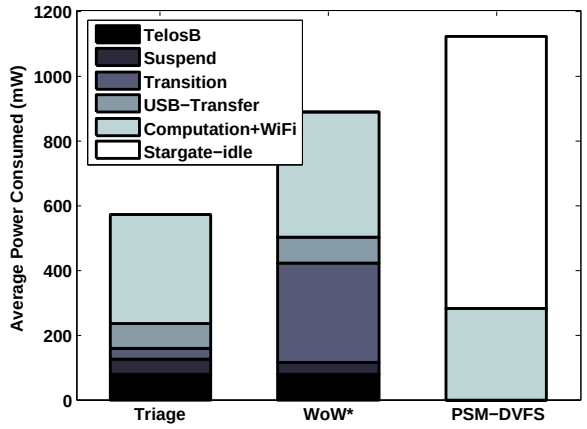

Figure 15: Break-down of the power consumed by independent components of Triage, WoW* and PSM-DVFS systems.

Several sensor network systems utilize a subset of the participating nodes as aggregaters, central processing nodes, or gateways [10]. This work can be classified into algorithms for networks of homogeneous devices and algorithms for networks of heterogeneous devices. In homogeneous systems such as Heed [31] and LEACH [12], the leader, or clusterhead, rotates among nodes in the network. The goal is to distribute the extra energy drain incurred by the leader. In heterogeneous systems, larger, more powerful nodes called microservers *herd* other smaller nodes [11]. Our work focuses on the latter scenario and addresses the need for a power-aware software and hardware architecture to reduce the energy drain on the resource-rich nodes.

8.2 Energy Management

Reducing the power consumption of mobile devices has been the subject of much research. Approaches include scaling the CPU voltage and frequency [8], managing wireless interface usage [1], turning off banks of RAM [13], or employing microsleep [4, 14]. In a larger device, such as a Stargate, these techniques still do not enable a power mode comparable to a Mote device. While these efforts target optimization for laptops and PDAs, our architecture targets microservers for wireless networks, and is designed to exploit a hardware platform with complementary components.

Papathanasiou and Scott made an observation similar to ours: batching work, or increasing idle periods, leads to greater energy efficiency [21]. However, the goal of the their work was to increase burstiness in laptop disk drives.

In our previous work, we designed a multi-tiered multi-radio architecture to design solar-powered energy efficient DTN routers [3]. The architecture uses a mobility prediction engine, and a lifetime scheduler to efficiently route network packets at minimal energy cost. Unlike Triage, which is an architecture to build general purpose microservers in a static sensor network setting, the throwbox architecture focused on application of a similar hardware architecture in a more mobile scenario. Although the design and algorithms were specific to disruption tolerant networking and cannot be applied to static sensor networks, the paper shows the efficacy of the hardware architecture in a mobile network scenario where contacts are sparse and short-lived.

The Wake-on-Wireless project (WoW) [26] proposes a hierarchy of devices for PDAs, including a low-power receiver that can wake the PDA. Our goal is similar to WoW, to reduce power consumption in battery powered devices. Their focus was solely on exploiting low-power radios, whereas Triage tackles the significantly broader problem of building energy-efficient, QoS-aware microservers.

Our Turducken system [27] employs multiple hardware tiers in the context of an always-on laptop system. The system designer predetermines the tasks to be executed on a tier. Unlike Triage, in Turducken task sharing is static and the system lacks a clean software architecture for automatic distribution of tasks among tiers. Moreover, Turducken system provides an always-on capability to laptops at minimal energy cost while Triage is an architecture for building a power efficient general purpose microserver in a static sensor network setting.

Narayanan et. al use a history-based scheme to predict the effect of application fidelity on resource consumption [20]. The predictors designed in the paper were specific to the platform and input data on which they are executed. Such predictors allows a system to learn over time, the behavior of an application and can provide feedback about its resource consumption. The paper uses the predictors in an operating system platform to monitor logs and predict how application resource consumption varies with fidelity.

8.3 Sensor Platforms

Recently, many embedded sensor platforms have emerged. These platforms span a broad spectrum of power requirements and functionality. A popular instance of sensor platforms is the family of Motes. These nodes are commercially available, widely used, and include the Crossbow MicaZ and Mica2Dot as well as the Telos node. All of these nodes consume peak power between 10-100mW and are tuned to be highly power efficient.

There are also several more capable but still very power-efficient sensor nodes such as the Yale XYZ [18]. This node has dynamic frequency scaling capability and can operate between 2MHz and 56MHz with a power consumption of up to 3x greater than a Mote at comparable clock speeds. Such intermediate platforms can be used as clusterheads in applications that have moderate computation requirements.

Our architecture targets resource-rich but power efficient sensor platforms that combine two processing elements—one small and one large. Such architectures have been used in other research efforts. The Stargate platform [30] incorporates a connector to a Mica or Telos Mote, but the intention was to provide a gateway for Mote radios, rather than to optimize the energy efficiency of the platform. The PASTA node is an architecture that combines a trip-wire board with a DSP processor together with a PXA processor [25], and the LEAP platform [7] integrates a higher-end processor-radio module (Intel PXA255 XScale running Linux) with a lower-end processor-radio module (TI MSP430). The mPlatform is another modular sensor platform which provides real time processing for requests on heterogeneous processors [17]. While these efforts employ hierarchical structures, they do not provide software architectures or algorithms for intelligently controlling the use of the hardware infrastructure.

9. CONCLUSIONS

This paper presents the design, implementation, and eval-

uation of Triage, an architecture for QoS-aware, energy-efficient microservers. Our work exploits hierarchical hardware with two connected platforms, one with high capability for executing a batch of tasks and one with high energy-efficiency while waiting for new tasks to arrive. A novel aspect of our work is our energy and QoS-optimized scheduler that uses extensive *in-situ* profiling capabilities to efficiently execute storage, communication and processing tasks. We demonstrate the use of two schedulers in Triage, an As-Late-As-Possible scheduler that optimizes for task deadlines, and a token-bucket based scheduler that optimizes for node lifetime. Our results show that Triage provides a 300% increase in microserver lifetime over existing systems and provides probabilistic quality of service guarantees.

While our Triage prototype demonstrates the potential for long-lived QoS-aware microservers, we believe that the ceiling is significantly higher than what we have demonstrated in this paper. First, we only considered an always-on tier-0 platform since efficient duty-cycling support is not yet available for the CC2420 radio on the Telos Mote. We believe that Triage can provide significantly more benefits if tier-0 were also duty-cycled, especially when requests are infrequent. For instance, we can reduce the tier-0 energy by a factor of 10 while increasing average latency by 1 second [24]. In such a scenario, a WoW-style system could be used to control the wakeups of tier-0 which can consequently wakeup tier-1 when required—this is something we aim to study as future work. Second, the hardware used in the current prototype is not ideal for a number of reasons including the large latency in waking the Stargate from suspension, its low bandwidth available to transfer data from the Telos to the Stargate. The large latency to wake the Stargate is the greatest limiting factor as it restricts the smallest latency requests that the microserver can support. As low-power platforms continue to evolve we expect that most of these bottlenecks will improve making Triage an even more useful and efficient system for untethered sensor deployments, mobile computing, and ubiquitous computing.

Acknowledgements

We thanks our anonymous reviewers and our shepherd Margaret Martonosi for their valuable feedback and comments. This material is based upon work supported by the National Science Foundation under awards CNS-0447877 and CNS-0546177, CNS-0520729, CNS-0519881, DUE-0416863, CNS-0615075. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

10. REFERENCES

[1] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning wireless network power management. In *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom'03)*, San Diego, CA, September 2003.

[2] R. Balani, S. Han, V. Raghunathan, and M.B.Srivastava. Remote Storage for Sensor Networks. UCLA-NESL-200504-09, UCLA, 2005.

[3] N. Banerjee, M. D. Corner, and B. N. Levine. An Energy-Efficient Architecture for DTN Throwboxes. In *Proceedings of Infocom*, May 2007.

[4] L. S. Brakmo, D. A. Wallach, and M. A. Viredaz. microSleep: A technique for reducing energy consumption in handheld devices. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (MobiSys'04)*, Boston, MA, June 2004.

[5] B. Burns, O. Brock, and B. N. Levine. MV routing and capacity building in disruption tolerant networks. In *Proceedings of IEEE Infocom 2005*, March 2005.

[6] Crossbow Technology Inc., San Jose, CA. *Stargate Developer's Guide*, Rev. A edition, September 2004. 7430-0317-12.

[7] K. Ho D. McIntire, B. Yip, A. Singh, W. Wu, and W. J. Kaiser. The Low Power Energy Aware Processing (LEAP) Embedded Networked Sensor System. Technical report, UCLA, Los Angeles, CA, 2005.

[8] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the Seventh ACM International Conference on Mobile Computing and Networking (MobiCom'01)*, Rome, Italy, July 2001.

[9] D. Gay, P. Levis, R. V. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.

[10] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems*, Baltimore, MD, 2004.

[11] R. Govindan, E. Kohler, D. Estrin, F. Bian, K. Chintalapudi, O. Gnawali, S. Rangwala, R. Gummadi, and T. Stathopoulos. Tenet: An architecture for tiered embedded networks. Technical Report TR-56, CENS, November 2005.

[12] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocols for wireless microsensor networks. In *Proceedings of the Hawaiian International Conference on Systems Science*, January 2000.

[13] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. In *Proceedings of USENIX Technical Conference*, San Antonio, TX, June 2003.

[14] N. Kamijoh, T. Inoue, C. M. Olsen, M. T. Raghunath, and C. Narayanaswami. Energy trade-offs in the IBM wristwatch computer. In *Proceedings Fifth International Symposium on Wearable Computers*, Zurich, Switzerland, October 2001.

[15] P. Kulkarni, D. Ganesan, and P. Shenoy. Senseye: A multi-tier camera sensor network. In *ACM Multimedia*, 2005.

[16] M. Li, D. Ganesan, and P. Shenoy. PRESTO: Feedback-driven data management in sensor networks. In *Proceedings of the 3nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2006)*, May 2006.

[17] D. Lymberopoulos, B. Priyantha, and F. Zhao. mPlatform: A Flexible and Efficient Architecture for Sharing Data in Stack-Based Sensor Network Platforms. In *MSR-TR-2006-142*, October 2006.

[18] D. Lymberopoulos and A. Savvides. XYZ: A motion-enabled, power aware sensor node platform for distributed sensor network applications. In *Proceedings of Information Processing in Sensor Networks (ISPN)*, Los Angeles, CA, April 2005.

[19] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Workshop on Wireless Sensor Networks and Applications*, Atlanta, GA, September 2002.

[20] D. Narayanan, J. Flinn, and M. Satyanarayanan. Using history to improve moble application adaptation. In *Proceedings of 3rd IEEE Workshop on Mobile Computing Systems*, December 2000.

[21] A. E. Papathanasiou and M. L. Scott. Energy efficiency through burstiness. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, Monterey, CA, October 2003.

[22] T. Pering, Y. Agarwal, R. Gupta, and R. Want. Coolspots: Reducing the power consumption of wireless mobile devices with multiple radio interfaces. In *Proceedings of Mobisys*, June 2006.

[23] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, April 2005.

[24] Joseph Polastre, Jason Hill, and David E. Culler. Versatile low power media access for wireless sensor networks. In *SenSys*, pages 95–107, 2004.

[25] B. Schott, M. Bajura, J. Czarnaski, J. Flidr, T. Tho, and L. Wang. A modular power-aware microsensor with 1000x dynamic power range. In *Proceedings of Information Processing in Sensor Networks (ISPN)*, Los Angeles, CA, April 2005.

[26] E. Shih, P. Bahl, and M. J. Sinclair. Wake on Wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the Eighth ACM Conference on Mobile Computing and Networking*, Atlanta, GA, September 2002.

[27] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of The Third International Conference on Mobile Systems, Applications, and Services (MobiSys '05)*, Seattle, WA, June 2005.

[28] J. Sorber, A. Kostadinov, M. Garber, M. D. Corner, and E. D. Berger. eFlux: A Language and Runtime System for Perpetual Mobile Systems. In *Technical Report 06-61, University of Massachusetts, Amherst*, December 2006.

[29] RA Walker and S Chaudhuri. Introduction to the scheduling problem. In *IEEE Design & Test of Computers*, 1995.

[30] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light. The personal server - changing the way we think about ubiquitous computing. In *Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing*, Goteborg, Sweden, September 2002.

[31] O. Younis and S. Fahmy. HEED: A hybrid, energy-efficient, distributed clustering approach for ad-hoc sensor networks. *IEEE Transactions on Mobile Computing*, 4(4), October 2004.