Triage: A Power-Aware Software Architecture for Tiered Microservers

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

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

†Department of Computer Science
Mt. Holyoke College, South Hadley, MA
srollins@mtholyoke.edu

Abstract—
Microservers are resource-rich sensor network nodes that perform complex tasks such as image processing, database query processing, localization, and target classification. While microservers can accomplish complex tasks, the required resources come at the cost of increased power consumption. To address this problem, a number of embedded platforms have employed a tiered architecture. Each tier is an independently operating subsystem containing its own memory, storage, wireless interface, and processor. Typically, such a hardware platform combines tiers with widely varying resources and power consumption characteristics. This presents a substantial opportunity to develop energy-efficient software systems; however, that opportunity has not yet been fully realized.

In this paper, we present the design of *Triage*, a software system for tiered microservers. The overarching goal of Triage is to reduce the amount of time a high-power tier must remain on by enabling a low-power tier to execute tasks on its behalf and batch work when possible. Our evaluation demonstrates that by using the low-power tier to batch file system reads and writes, we achieve a battery lifetime *four times* longer than a system that wakes the high-power tier for every operation. Further, using the low-power tier to execute smaller image processing tasks can increase the system lifetime by up to *four times*.

I. Introduction

Untethered sensor network deployments use battery-powered nodes to gather, process, and store streams of sensed data. These deployments employ a heterogeneous collection of nodes, sensors, and applications. Example applications include video processing [4], acoustic detection and localization [14], as well as general signal processing [22]. The sensors needed for these applications produce data that requires online processing and in-network storage [11] for event correlation and post-processing.

Typically, designers employ low-power nodes with limited processing and storage to read input from the sensor, aggregate data, or store it for future use. This enables inexpensive deployments over a wide area. While small-sized nodes, such as motes, work well at the edges of the network, they are poorly suited for the demands of performing complex signal processing tasks, storing large amounts of persistent data, or running a full database with query processing.

In response, recent projects have proposed the use of more powerful nodes that act as *microservers* within the network [37]. In one such scenario, depicted in Figure 1, small-sized wireless nodes gather sensor data, and feed them back to a local microserver for processing and storage. Video data, acoustic data, and seismic information are then compressed, processed, and stored in an in-network storage system or database. The set of microservers form a powerful back-bone for the network, communicating directly or through routing nodes to disseminate, aggregate, or query sensed data.

Remote sensing applications, such as James Reserve and Great Duck Island [26], depend on untethered, battery-powered, microservers to provide in-network storage and processing. While there are fewer microservers than sensor nodes in remote locations, power is equally inaccessible to all nodes. It is crucial that the microserver is energy efficient, as increased energy consumption translates into a larger battery, one with a shorter life, or large and intrusive solar arrays [26].

Unfortunately, high-power platforms are less efficient than low-power designs for lightweight tasks such as storing incoming data or processing small amounts of data. One method to ensure that a microserver operates with minimal power consumption is the use of a *tiered* hard-

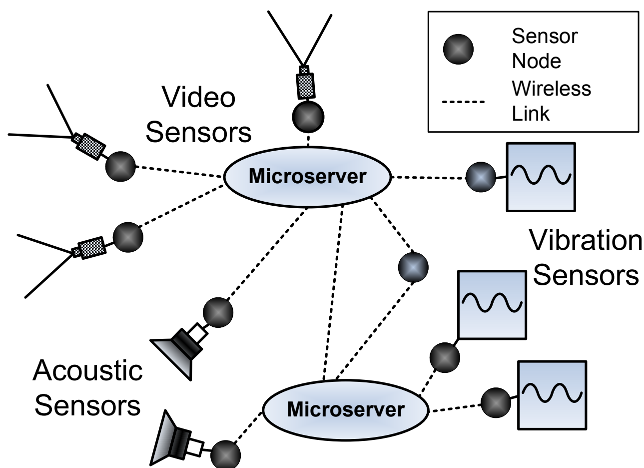

Fig. 1. Microserver Deployment

ware platform. This platform contains two or more distinct *subsystems* operating at different power points. The less powerful subsystem, or lower tier, can handle lightweight tasks while the upper tier remains in a power saving mode. This approach is collectively referred to as Hierarchical Power Management (HPM) [36].

This tiered hardware design presents an opportunity to substantially lower the power consumption of a microserver through intelligent software control. In this paper, we present the design of *Triage*, a software architecture for microservers. Triage reduces the amount of time a high-power tier must remain on by enabling a low-power tier to execute tasks and batch work. Triage also balances responsiveness with energy consumption. Specifically, we propose the use of *surrogate* services that run on the low-power tier. While the high-power tier is in a power-saving mode, the surrogates act on its behalf. A *dispatcher*, running on the low-power tier, determines where and when tasks should be executed in order to ensure a low system-wide energy cost.

To support the kinds of applications highlighted in the introduction, we have created a working prototype that runs on a slightly modified Stargate platform. We provide several surrogate services, including a generic execution service, an image processing service, and an in-network storage service. We evaluate the prototype's performance against current designs using two applications: a storage server and an image processing server. By using the low-power tier to batch storage reads and writes, our system achieves a battery lifetime *four times* longer than a system that wakes the high-power tier for every operation. Further, using the low-power tier to process small images, while sending the large ones to the high-power tier, Triage can increase the system lifetime by up to *four times*.

II. Background

Our goal is to develop a software control architecture to support Hierarchical Power Management (HPM) [36]. HPM refers to the technique of using several tightly coupled hardware platforms as a single integrated system that is optimized for energy efficiency. This section describes the characteristics of this kind of hardware system.

Generally, platforms with more resources (e.g., processing and memory) can complete tasks faster as well as those with higher resource requirements. However, resources come at the cost of greater power requirements. Tiered hardware platforms address this dichotomy by combining two or more tightly coupled but independently operating embedded subsystems, as shown in Figure 2. The upper tiers are strictly more capable and power-hungry than lower tiers. This means that any task a lower-tier system can do, a higher-tier system can also perform. Platforms such as the Stargate [38, 6], Turducken [36], and the PASTA sensor node [34] employ this model.

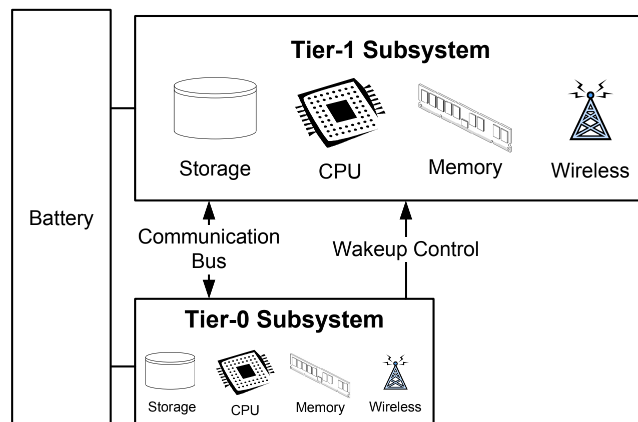

Fig. 2. Hierarchical Power Management

Tiers are tightly coupled and directly communicate over a wired link. This enables a lower tier to trigger the wakeup of a higher tier when necessary. Also, while a particular tier is not in use, it can be shutdown, suspended, or hibernated to save power. We assume that the tiers draw power from a single battery. Because both tiers must have power for the platform to work, using a single power source multiplexes the power draw optimally.

The HPM technique is particularly effective for a platform containing subsystems that are separated in power consumption and capabilities by an order of magnitude or more. For instance, a Stargate subsystem draws 40 times the power of an Atmel-based mote and computes at 67

times the clock rate and four times the bus width—this is the combination we use in our implementation. While the node was originally intended to act as a gateway, it provides a readily available HPM test platform.

III. Software Architecture

Existing hardware provides the foundation for applying the HPM technique, but the hardware alone does not provide energy-efficient operation. Our goal is to design a dynamic software architecture that *dispatches* network-supplied *tasks* to the tier that can most *efficiently* complete each task given a set of *constraints*. A task is any processing, storage, or communications job that the microserver needs to perform. Efficiency is determined based on the amount of energy it requires for a given tier to execute the task. Any measurement of task energy efficiency must also consider any overhead associated with waking a tier and transferring the task. Constraints may refer to the latency of completion of the task or the resource requirements of the task. Dispatching or routing of tasks is crucial in providing a long lasting but responsive microserver.

A. Design Goals

The Triage design is based on the following goals:

- **Dispatch tasks to meet efficiency constraints.** A task should be routed to the tier that can accomplish it most efficiently. Associated with each task is a set of parameters detailing the predicted energy and time required to complete the task on a given tier, the capabilities required of a tier executing the task, and any constraint on the latency of completing the task. Using this information, the architecture can make a routing decision that meets the specified constraints in the most efficient manner possible.

- **Aggregate and defer work when possible.** Deferring work—a traditional OS technique—allows the system to aggregate tasks so that the higher tier processes them en masse. As we show in our evaluation, waking the higher tier is a critical factor in the lifetime of the system. Deferring work as long as possible improves system lifetime by amortizing the wakeup cost over a set of batched tasks.

- **Optimize tasks when possible.** Performing optimization on the set of batched tasks allows the system to hold more tasks in limited storage as well as enables the system to fulfill requests directly from the batch. For instance, if the lower tier receives an updated value for a reading from a sensor node, it can cancel any previous writes for that value. This technique can reduce the number of tasks that must be passed to the higher tier and enable longer suspend/off periods for the higher tier. In some cases, the lower tier may also be able to complete a task without passing it to the higher tier. For example, if the lower tier receives a database query for information that it has cached, it may respond without waking the higher tier.

B. Architectural Components

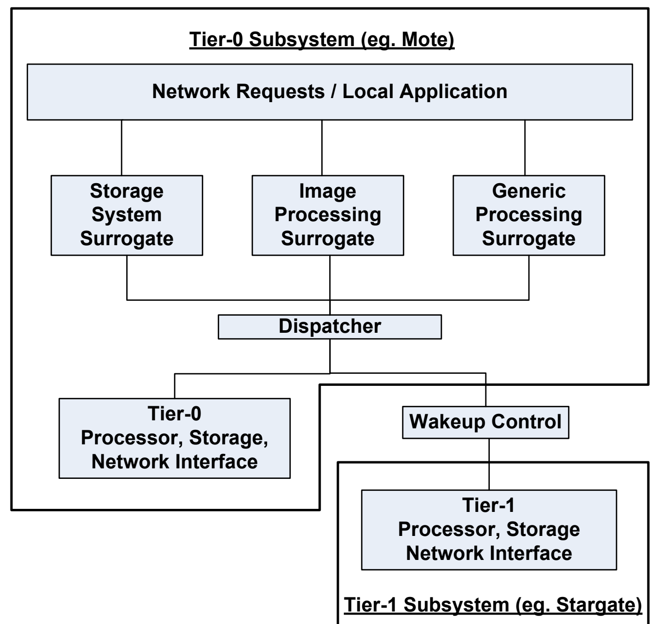

Fig. 3. Triage Architecture

Our Triage software architecture, shown in Figure 3, provides a set of operating system extensions to enable efficient use of a tiered microserver. We have designed our mechanisms such that they can lie outside of or within the operating system of each tier. Current trends in sensor network operating systems blur the line between applications and modules, and we have made no effort to resolve that difference.

The locus of control in Triage lies in the least powerful tier of the platform—tier 0. When a task arrives at tier-0, it is passed to a *surrogate* responsible for that type of task. The surrogate determines whether a task can be immediately processed, or defers the work by writing it to a log. This log is held in the local storage of tier-0. The surrogate also logs several extra *task parameters* to aid in the routing decision. A *dispatcher* makes decisions on aggregating requests and waking the more powerful subsystem: tier-1. Each of these elements are detailed below.

B.1 Task Parameters

To assist the dispatcher in making routing decisions, the surrogate stores each task with a set of parameters:

- **Responsiveness constraint:** The responsiveness constraint describes a hard deadline for task completion.
- **Responsiveness prediction:** The responsiveness prediction estimates how long the task will take to execute on each tier.
- **Capability constraint:** The capability constraint describes the tiers that possess the resources, for example memory or storage, to complete the task. This constraint prevents the system from assigning to tier-0 large tasks that it cannot complete.
- **Efficiency prediction:** The efficiency prediction specifies the amount of energy, in joules, that the task is anticipated to consume on each tier of the system.

The responsiveness constraint helps the dispatcher determine how long a task can be deferred while the efficiency prediction, capability constraint, and responsiveness prediction help the dispatcher to determine where the task should be performed.

The surrogates must generate the responsiveness prediction, capability constraint, and efficiency prediction. One approach is to use a model for each kind of task including execution time on each tier, the power draw of that tier during that task, and the size of input that the tier can handle. For many sensor network tasks, such as signal processing and storage system writes, simple models based on the size of the input parameters can be formed a-priori—we give an example of one such model in our discussion of the image processing surrogate. The responsiveness constraint is optionally supplied from network nodes requesting service from the microserver. If no responsiveness constraint is included, the microserver will delay its execution as long as possible.

B.2 Surrogates

The surrogates act on behalf of tier-1. When possible, they execute tasks that can be more efficiently completed on tier-0 enabling tier-1 to remain in a low-power state. We envision a host of surrogates: a generic processing surrogate, an image processing surrogate, a storage system surrogate, a database surrogate, and a network routing surrogate. These components provide standard interfaces to applications or services running on the tier-0 system, such as a reduced version of the UNIX file system interface.

Each surrogate manages a *log* of pending tasks and determines task parameters for incoming tasks. A surrogate may also directly execute or optimize tasks based upon any semantic information it has about the tasks contained in the log. When a surrogate receives a task, it first determines whether it can execute the task based on entries cached in the log. This provides low latency for operations that have high degrees of temporal locality. For instance, in the case of a storage system, a read may be serviced by previous writes found in the tier-0 log. If a task cannot be serviced using cached log entries, the surrogate generates the task parameters and inserts the task and its parameters in the log. After logging the task, the surrogate notifies the the dispatcher that there is new work in the log.

Managing the size of the log is critical in extending the lifetime of the microserver. Once the log fills, tier-0 has no choice but to wake tier-1 and truncate the log. To reduce the size of the log, thus deferring the wakeup of tier-1, surrogates periodically perform log cancellation optimizations. For example, if the log contains a database insertion and a new insertion overwrites it, the previous insertion can be removed from the log.

B.3 Dispatcher

The dispatcher makes routing decisions that specify when and where to execute tasks logged by the surrogates. This decision is based on the deadline for the task, the predicted amount of time it will take to execute the task, the predicted energy cost of executing the task on each tier, the amount of time to transfer the task's parameters and results if it is executed on tier-1, and the cost of waking tier-1 if necessary.

When the dispatcher is notified that a new task has been logged, it first determines if the task can be executed by tier-0 at the lowest energy cost. If so, it immediately begins execution of the task—there is no benefit in delaying its execution. Otherwise, the dispatcher defers the execution decision until the deadline is imminent.

Responsiveness constraints are considered hard deadlines, but rely on a correct estimate of task completion time. The dispatcher must subtract the estimates of completion and transfer time from the deadline to determine the actual time the task will finish. Additionally, the dispatcher assumes that tasks and transfer of parameters are completed serially, with no concurrency between jobs. Although the dispatcher has assumed serial transfer of parameters, many surrogates are capable of pipelining parameters and tasks. For instance, the storage system surrogate can begin writing data to disk as soon as the system boots, and the leading edge of the log arrives.

Delaying the execution as long as possible is consistent with our principle to defer work as long as possible. This

increases the chances that the task will be canceled through a log optimization, and maximizes the efficiency gains of batching work. However, because the space for logging work is limited, the dispatcher may be forced to truncate the log even if a deadline is not imminent. Moreover, many tasks, such as updates to the tier-1 storage system, must be executed on tier-1, and the dispatcher has no choice but to wake it to begin execution.

When the dispatcher determines that it must wake tier-1, it does so using the control interface and sends the tasks and parameters to a daemon running on tier-1. Tier-1 receives the data, performs the task, sends the results to tier-0, and returns to a low-power state.

IV. Example Surrogates

We present three example surrogates to demonstrate the wide applicability of Triage. The first surrogate is a general processing surrogate. It exposes a very limited interface, and only provides the benefits of batching at tier-0. The second surrogate is an image processing surrogate which is able to take advantage of semantic information that is not available to the generic processing surrogate. This information allows tight estimates of computation time and energy consumption which result in more efficient task dispatching. The third surrogate is a storage system. The storage surrogate is an example of a surrogate with semantic knowledge, and the ability to optimize and cache results.

A. Generic Processing Surrogate

The generic processing surrogate provides an RPC-like interface, allowing the programmer to specify any task in much the same way as a local procedure call. For instance, a programmer can provide an RPC that compresses an audio stream, or performs an FFT.

In order to meet timeliness constraints, route tasks, and effectively conserve power, a surrogate must be able to predict the time and energy required to execute each task. Predicting the execution time of a generic function on a given input is an unsolvable problem without additional function-level information. The client may have additional information about the requirements of the call, but the microserver provides no information to clients about its own energy efficiency or capabilities. Due to these limitations, clients can only send tasks that can be computed on tier-1 and limits Triage to batching only—it cannot provide routing or timeliness guarantees.

When a remote node sends data to the microserver for processing, tier-0 logs a remote procedure call, the data, and the task parameters. When tier-1 finally executes the RPC, the results are written back into the log and are delivered to the requesting node. Since the surrogate possesses no semantic knowledge of the individual tasks, it cannot perform any useful optimizations on the work queue. Even caching results from identical RPC calls is not possible without severely restricting the kinds of jobs being executed.

B. Image Processing Surrogate

The image processing surrogate is capable of detecting objects within an image using a standard kmeans algorithm. The object detection task can be performed on either tier, and time and energy estimates are used to route the task for execution on the proper tier. A full image processing surrogate would provide many other functions, such as compression, object recognition, classification, and segmentation.

In order to accurately estimate computation time we empirically developed a model for the kmeans function based on image size and the maximum number of objects being detected. We observed that for a fixed image size the compute time grows linearly with the maximum number of objects. However, for a fixed number of objects the compute time grows quadratically in the total image size (i.e. pixel count). Based on these two observations, we decided on a simple model for estimating time in the following form, where T is the compute time, S is the image size in pixels, and N_o is the maximum number of objects to detect:

$$T = c_1(S - c_2)^2 + c_3 N_o + c_4 \tag{1}$$

We ran this computation on many different images of different sizes and with different values for N_o in order to appropriately set the constants $c_1, \ldots, c_4$ for both tiers. Note that this is only one possible model that could be used; however, for any such model, the required accuracy is determined by the cost of missed deadlines. If timeliness constraints are hard real-time deadlines, then a more accurate, or at least more conservative model is needed. In building this surrogate we assumed a soft real-time policy, allowing a few minor violations of timeliness constraints due to small inaccuracies in our prediction model.

C. Storage System Surrogate

One of the primary motivations for deploying a microserver is to provide data storage resources far beyond that of a smaller sensor node. We enable this through a storage system surrogate. Its interface has been greatly simplified from UNIX-style file systems and does not currently provide directories, lookup, or partial writes—all files must be written in their entirety. Not supporting partial writes is somewhat limiting, however, it is similar to

a block style interface, where all blocks must be written in entirety. Full UNIX-style semantics are unnecessary to demonstrate the utility of this surrogate; however, we plan to expand the interface in future Triage implementations. The full contents of the storage system are stored on tier-1's storage system. In the case of the Stargate platform, this is 32MB of flash memory.

Since Triage aggressively suspends or powers down tier-1, the full storage system is typically unavailable. In order to provide more energy-efficient data availability, the lower tier, holds a log that is a subset of the storage system. It contains the most recent write requests, read requests, metadata updates, and also a cache of recently read results. The use of logging is similar to a Log Structured File System [32] and the split of the log and full storage system is similar to those found in distributed file systems such as Coda [33].

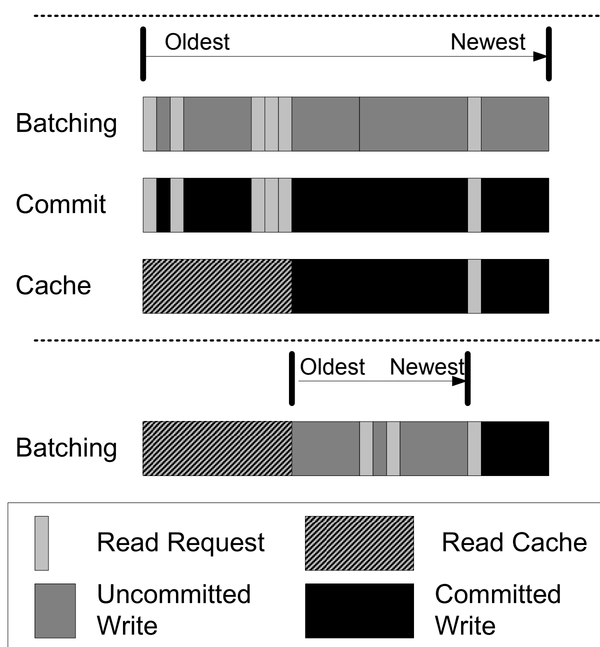

Fig. 4. Storage System Log Management

Providing efficiency estimates is much easier for this surrogate than for the image processing surrogate. As all data is eventually committed to tier-1, for writes it is unnecessary for the surrogate to provide any estimates of energy use or responsiveness. Even though tier-0 does not immediately commit updates to the full high-power storage system, the system still provides write-read consistency, as the written data still exists in the log. Any subsequent reads on the same data will be fulfilled by previous writes in the log.

In the case of reads, remote nodes can optionally provide a timeliness criteria to this surrogate. This guarantees that the result of the read request will be fulfilled by a certain time. Without this parameter, Triage will not provide the results of the read until its log fills or until the timeliness constraint of another task forces a commit. The storage system service has an optional timeliness parameter that serves as a default maximum to bound responsiveness.

When a remote node sends a write request to the microserver, it logs the write request and data. Similarly for read and delete requests, the request is logged. For each read request, the system consults a log index to determine whether or not it can provide the data immediately from the log. If it cannot, then the read is delayed as long as its timeliness constraints will allow.

For each request the surrogate attempts to optimize the log. A delete request cancels any previous writes in the log. It is not necessary to look for intervening reads, as partial writes are not allowed, and those read requests would have already been fulfilled by a logged write.

When the log finally fills and no further optimizations are possible, the log must be committed to tier-1. Triage wakes the tier-1 subsystem, and begins sending the log over the local bus. The higher tier commits all writes and deletes, as well as returning the results of read requests. The results of all operations are written to the log on the lower-tier, and the higher tier goes back to sleep. The results of read requests, and the previous set of writes remain in the log, and serve as a cache for future requests.

Cached storage data is managed using an approximation of the *least recently used* eviction policy. For writes, the time of use is considered to be when the microserver received the request—writes can be considered completed when Triage writes them to its stable storage. For reads, the choice is less clear: we could chose the time that the read request arrives, or the time it is completed by the system. Without a clear disadvantage, we chose that latter, as it is more convenient for avoiding fragmentation issues. To see why refer to the process demonstrated in Figure 4. During the batching phase, the log accumulates read requests along with uncommitted writes. During the commit phase Triage wakes tier-1 and commits all of the writes. When tier-1 returns the read results, they are also written into the log as a cache. The read itself just occurred, however the actual time of the request is interspersed with the recent set of writes. In light of the severe resource and computation constraints imposed by the lowest tier, and our preference to avoid fragmentation issues, we chose the simpler policy that favors these reads as "most recent". Read results are therefore written into the cache immediately preceding the most recent batch of writes. As space becomes scarce, the oldest part of the cache is the first to be overwritten by new requests. However, for a read re-

quest that hits in the cache, no effort is made to rearrange the flash to match this more recently used item—to do so would require several copy operations that the mote does not necessarily have space to complete.

V. Implementation

In order to evaluate our approach we have implemented a working prototype of our Triage architecture. This prototype, built for existing HPM hardware, consists of a dispatcher and three surrogates.

A. Prototype Hardware

We built our prototype on a hardware platform consisting of a slightly modified Crossbow Stargate (tier-1) and a MicaZ mote (tier-0). This hardware platform was chosen because it is easily programmable, well supported, and runs Linux, making available a broad range of software tools and services. While this platform was not originally designed for HPM, we made only minor modifications to the hardware to support it.

In order to support our software architecture, we decoupled the power control of the mote and Stargate, allowing them to operate independently; we added new control interfaces to the Stargate for waking, suspending and powering down the Stargate from the mote; and we had to unsolder a pin on the Stargate which was preventing the mote from accessing its flash memory while connected to the Stargate.

There are several methods for putting tier-1 into a low-power state. Tier-1 can be suspended (memory refreshed), hibernated (memory written to flash), or shutdown (memory not saved). The choice between these states depends on the length of time tier-1 is expected to be in the low-power state, and we consider this issue orthogonal to our work. When suspended the Stargate draws roughly 160mW of power as compared to other PXA-based platforms that draw less than 40mW in suspension. This overhead unfortunately makes suspension far too expensive to use. Even at 40mW, this is roughly equivalent to the amount of power normally drawn by the mote while executing. Regardless, our system is designed to keep tier-1 in a low-power state as long as possible, and except for heavy workloads, full shutdown is a better option.

This hardware platform is not completely ideal for Triage for two other reasons. The main limitations that impact energy efficiency are the serial transfer speeds and the power supply for the mote. First, the two devices communicate over a serial line which is limited to 57600 bps. As a result, transferring 512 KB of batched work along with protocol overhead takes more than 1 minute and wastes a great deal of energy while blocked on serial I/O. Improved HPM hardware platforms would allow faster transfer of tasks between tiers, either with a high-speed bus or by using shared memory. Second, when supplying power to the mote through the on-board connector, the mote draws roughly 160mW. This is 4 times the power the mote draws when powered by its own power supply. Unfortunately, supplying it from its own power supply is not an option as the mote is improperly isolated from the Stargate board. We are currently working on resolving this issue. In the evaluation section we present results that quantify the effect of each on energy efficiency.

B. Dispatcher and Surrogates

As part of this prototype, we implemented a dispatcher and three surrogates: generic processing, image processing, and storage system. Each surrogate exposes specific interfaces to the application, manages its work queue, and models the complexity and requirements of its tasks. Each of these system components were implemented according to the design described previously.

We implemented these components as TinyOS modules written in nesC [13]. Surrogate queues are stored and managed on the mote's measurement flash using the Matchbox filesystem [12]. The dispatcher and generic surrogate comprise 1,500 lines of nesC code, while the storage and image processing surrogates consist of roughly 2,000 and 500 lines of code respectively. We also implemented an execution engine which runs on the Stargate and executes tasks when they are received from the mote.

VI. Evaluation

The main purpose of Triage is to enable a battery- powered microserver to consume as little energy as possible without sacrificing responsiveness constraints.

A. Applications

We have developed two applications in order to demonstrate the benefits of our approach: a in-network storage server and an image processing server. These applications are representative of types of applications which are typically deployed on a microserver.

An in-network storage server has many advantages in a sensor network. It facilitates the sharing and aggregation of data across multiple sensor nodes and provides a simple solution to the storage limitations which are common to low-power sensors. This application services network requests to read from and write to the Stargate's filesystem. Requests are serviced using the storage system surrogate.

Image processing addresses another reason for deploying a microserver in a sensor network: computational limitations. In this application the image processing server

receives images and is responsible for classifying them using a k-means classification algorithm. Whenever an image arrives it is stored using the storage system surrogate and then processed using the image processing surrogate. This demonstrates how multiple surrogates can be used in concert by a single application.

B. Methodology

In each of our experiments, we measure the power consumed by the system using the method shown in Figure 5. We collect power traces by measuring the current that the system pulls across a 1-Ohm resistor. These measurements are taken using an Agilent 54621D Oscilloscope tracing the voltage drop across the resistor.

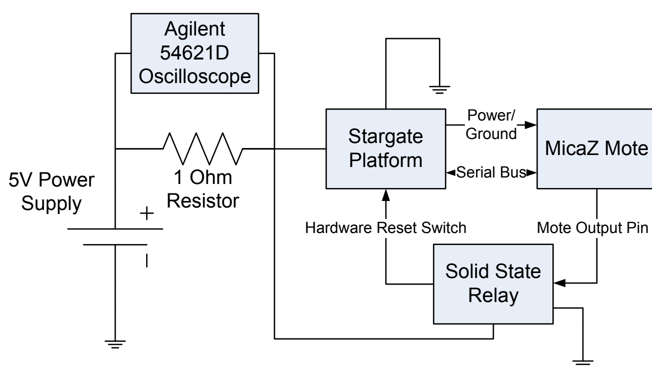

Fig. 5. Experimental Setup

This measurement approach is simple and accurate, but limits the size of traces that we can collect. This limitation is not inherent to the measurement method, but is due to a limitation of our oscilloscope, which can only record traces of up to 8 minutes.

We address this problem using time dilation. As shown in the sample trace in Figure 6, phase-specific power consumption patterns make it is easy to distinguish between the various phases of execution. For example, there is always a brief drop in power consumption at the end of the booting phase before it begins executing its first task. We have observed from our experiments that both the time between waking tier-1 and the time tier-1 spends executing tasks during each waking period are linear in the amount of flash being used on tier-0. This allows us to use only a portion of tier-0's flash storage, for experiments that would require longer than 8 minutes of batching to fill the available storage space. We then scale the batching phase and execution phase times to what they would be if the entire storage space was being used. The boot-up and shutdown time remain constant, since they do not change with the amount of data being processed.

For each experiment, we measure the average power consumed by the entire system during the batching and execution phases. These average power values are used when dilating these phases, and to compute the average power draw for the entire system. For example, in Figure 6 if the batching phase and the execution phase have average power consumptions P_b and P_e respectively and run for t_b and t_e seconds, the average power over the the total time would be given by $\frac{P_b \cdot t_b + P_e \cdot t_e}{t_e + t_b}$.

For each system we compare in our graphs, we measure the average power draw of the system under three different workloads. These measured values are shown with an "X" on the graphs. From these results, we use extrapolation to project system power consumption for a wide range of additional points. These projections are shown as lines along with the measured results.

We compare three approaches. The simplest approach uses tier-0 as a network interface (NIC), which listens for incoming tasks and forwards them immediately to tier-1 for execution. We implemented two variants of this NIC approach. The first powers down tier-1 in between task arrivals, and the second only puts tier-1 into a suspended state, which draws some power but allows for more rapid activation of tier-1. The other two approaches are the Triage system without log optimization, which uses only batching and routing, and the full Triage system with log optimizations.

As we previously described, the Stargate fails to appropriately isolate the two tiers from each other, which results in tier-0 drawing 4.5 times more power than it does when powered separately. To more accurately show the potential of our system, we estimate what our results would be if the power supply of the two tiers were properly isolated from one another. Such a system would draw the same power as a standalone mote when tier-1 is powered down. In our figures, these adjusted results, labeled Triage*, are presented alongside our measured results.

C. Image Processing Server

The purpose of our first set of experiments is to demonstrate the benefits of batching and routing. In each experiment, images arrive at the image processing server on tier-0 and are either processed locally or delegated to tier-1.

In the first experiment blocks of image frames arrive at the image processing server at different rates. Each image block processed has a size of 200 bytes. K-means classification of 200 byte blocks requires more memory than is available on tier-0. Therefore, these image blocks are batched and executed on tier-1 when tier-0 runs out of storage space. In this experiment tasks were not given responsiveness constraints.

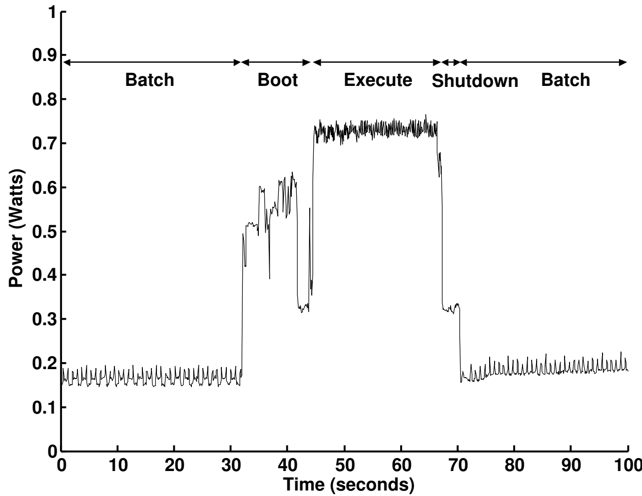

Fig. 6. A Sample Trace showing the various phases of the execution in an experiment

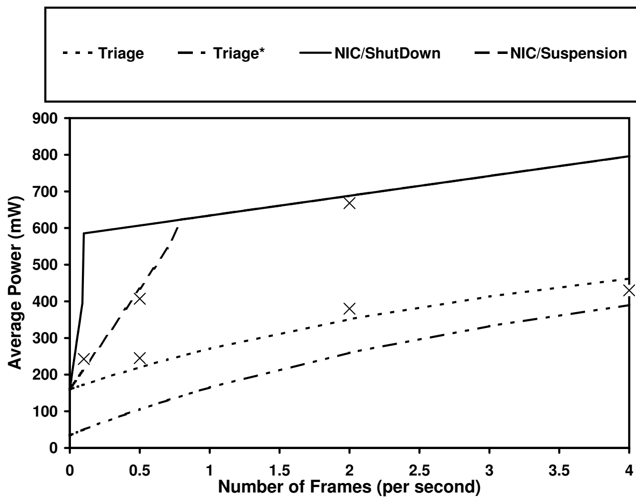

Fig. 7. The power consumed by the image processing server shown with respect to frame arrival rate. Triage consumes 4 times less power than the NIC.

The results of this experiment are shown in Figure 7. This shows the benefits that result from using batching in a tiered microserver. Without any additional routing or log optimizations, the Triage system consumes as little as 25% as much power as the NIC system. Triage achieves this energy savings by amortizing the cost of waking tier-1 across many processing tasks. Also, the Triage* numbers indicate that the potential gain is even greater, achieving as much as a 85% reduction in power consumption over the NIC approach. The graph also shows that if the frame blocks arriving per second is small it is more advisable to suspend the tier-1 system rather than powering it off. This is because for smaller frame arrival rate, we boot up the system more often and the overhead of the bootup is large. However, when the frame arrival rate is large tier-1 saturates and should always remain on to process the requests.

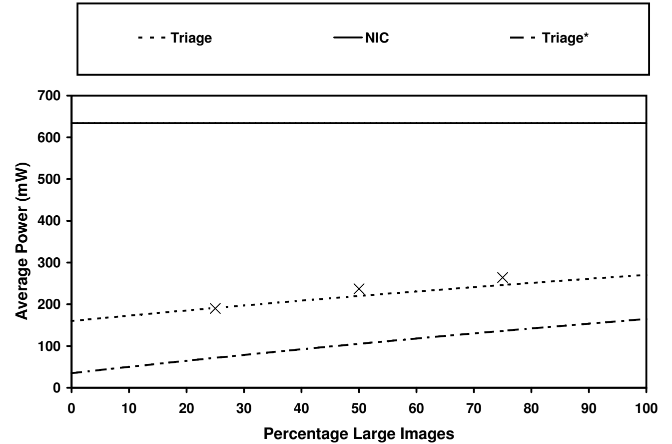

Fig. 8. The power consumed by the Image Processing Server is shown with respect to percentage of large frames in the workload. Triage is able to conserve energy by routing small images to tier-0 where they can be processed more efficiently.

The second experiment demonstrates the benefits of the dispatcher by introducing smaller image blocks (100 bytes) which can be processed more efficiently on tier-0 than tier-1. We fix the frame arrival rate at 1 frame/sec and vary the fraction of the workload that is made up of these small images. As in the previous experiment, processing tasks are not given responsiveness deadlines.

The results of this experiment are shown in Figure 8. This figure demonstrates the potential benefits that can be achieved using routing in addition to batching in Triage. Dynamically routing tasks to the most energy efficient tier results in as much as a 60% decrease in average power consumed, and a 65% reduction when 50% of the images are more efficiently processed on tier-0. If the tiers are appropriately isolated from each other, the power draw reduction increases to 85%. The degree to which routing will reduce energy consumption clearly depends on the system workload.

In our third experiment, we consider the effect of responsiveness constraints. We vary the deadlines for image processing tasks for a fixed frame rate of 1 large frame/second. Note, that we do not use any time dilation for this experiment.

Figure 9 presents the results of this experiment. This figure shows how application-supplied responsiveness constraints work in opposition to the power reductions achieved using batching and routing. If task results are needed immediately, Triage must constantly power tier-1,

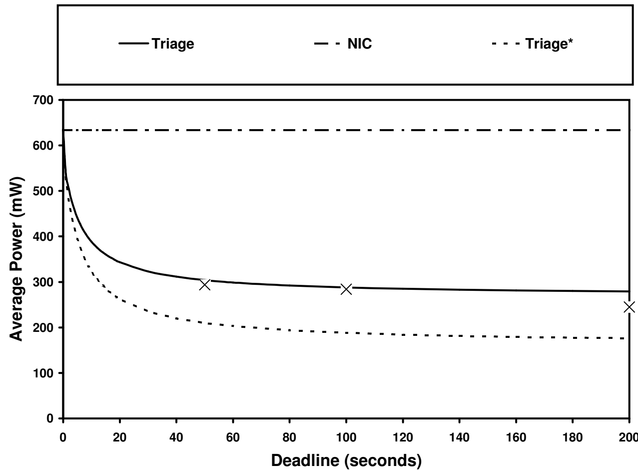

Fig. 9. The power consumed by the Image Processing Server is shown with respect to the length of task deadlines. Triage consumes 2 times less power than a NIC for smaller deadlines.

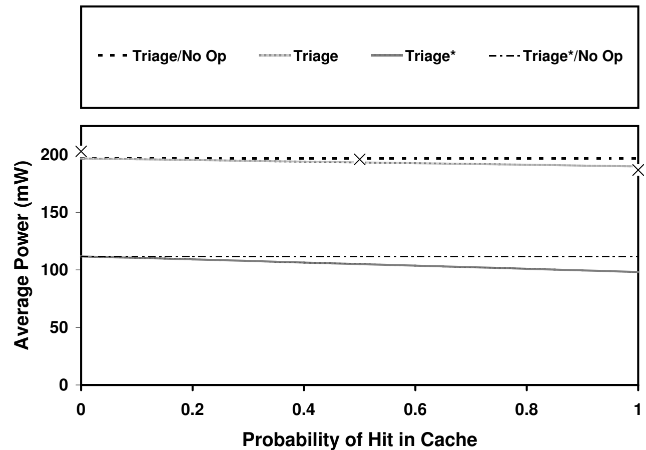

Fig. 10. The power consumed by the In-Network Storage Server is shown with respect to the increasing probability of read-hits in the cache. Triage saves 6% power over Triage with no optimization

and no power is conserved over the NIC system. Fortunately, we also see that if the application is willing to tolerate a 60 second delay, we get a 53% reduction in power draw over a NIC.

D. In-Network Storage Server

Our second experiment demonstrates how caching and log optimization in Triage result in additional energy savings. We vary the locality of storage server accesses for a fixed server load. In this experiment, the storage server services random reads from and writes, at a constant rate of 8 transactions per second, to a 20 MB data store located on tier-1.

The requests are not given any deadlines. It is also important to note that for this experiment Write-Write optimizations are not used. The reason for this is that due to implementation details of the Matchbox filesystem, this optimization takes $300ms$ to replace a single write in the log. As a result, we cannot currently support Write-Write optimizations for workloads of more than 2 transactions per second. We are currently looking into the ELF filesystem as a possible alternative to Matchbox in expectation of higher throughput.

The results of this experiment are shown in Figure 10. When the workload has higher locality, Triage is able to reduce the power consumption of the system up to 6%. This improvement is the result of using Write-Read optimizations only. We expect that adding Write-Write optimizations would provide additional benefits of larger magnitude since removing a write from the log saves more space than a read. Moreover, if the throughput of the filesystem is increased, larger number of optimizations would be possible which leads to larger power savings.

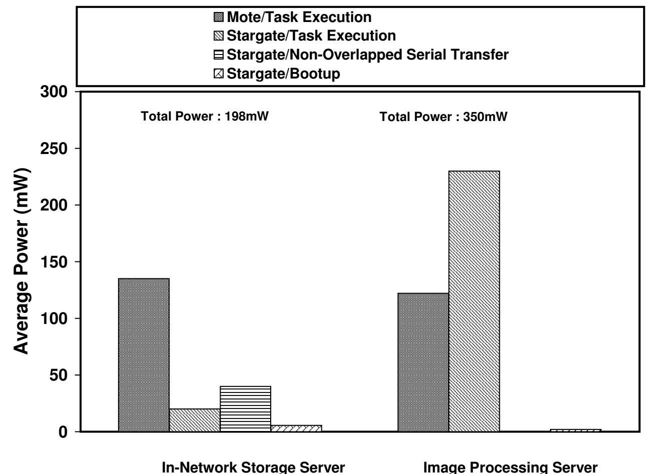

Fig. 11. The average power consumption is shown for different components of the Triage system.

E. Component Power Consumption

Finally, we examine the power consumption of the individual activities of the Triage system. Figure 11 shows this comparison for the image processing server, with a frame arrival rate of 2 frames/second, and the in-network storage server, processing 8 transactions/second. This figure shows the fundamental differences of these two applications. For the in-network storage server, the majority of the power consumed is by the mote (tier-0) and the transfer of data between tiers. The energy used to transfer task

data could be greatly reduced by using a higher speed data bus or a shared memory. The image processing server's tasks are much more computationally intensive resulting in a large portion of the average power consumed in processing the images on tier-1. Both of these applications show a significant power draw by tier-0. This power consumed by the mote has been discussed previously, and we hope to be able to modify our hardware to reduce this by at least a factor of 3. We also see that the time to boot the Stargate is not a significant factor in the average system power draw.

To express our results in terms of battery lifetime, if we use a Lithium-ion laptop battery of capacity $47.5Wh$, Triage* would last 20 days as an image processing server that receives two frames/second. However, if we duty cycle Triage* $\frac{9}{10}$ of the time , it can last up to 6.5 months as compared to a NIC system which would last for 33 days under similar conditions.

VII. Related Work

The design and implementation of Triage draws from several related research areas, which we survey here.

A. Microservers and Clustering

Several sensor network systems utilize a subset of the participating nodes as aggregaters, central processing nodes, or gateways [15]. 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 [41], LEACH [17], and the system proposed by Bandyopadhyay and Coyle [3], 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 [37]. Our work focuses on the latter scenario and addresses the need for a power-aware software architecture to reduce the energy drain on the resource-rich nodes.

B. Disconnected Systems

Triage is similar to many mobile systems in that parts of the system can become disconnected when suspended, hibernated, or powered down. Several mobile systems have addressed disconnection and lack of availability between participating nodes. Examples include file systems such as Coda [33] and Ficus [31], databases such as Bayou [8, 29] and DBmate [30], remote execution systems, such as Spectra [10] and Chroma [2], and general toolkits such as Rover [19]. Many of the techniques found in these systems, such as logging and caching, strongly influenced our design. In particular, we used the queued RPC mechanism from Rover as a basic building block. However, Triage differs from traditional mobile systems. First, with Triage there is a new element of control: the "client" (the less powerful system) can directly control when connections and disconnections occur to the "server" (the more powerful system). Second, the less powerful system is more resource-constrained than a typical mobile laptop—our lowest tier only contains 4kB of program memory. Additionally, we have integrated the competing concerns of quality of service and energy in our design criteria.

C. File Systems

Logging file systems have been proposed both for hard disk drives, such as the Log-Structured File System [32], as well as flash memory, such as ELF [7] JFFS2 [40], and Matchbox [12]. In our prototype, tier-0 uses the Matchbox file system and tier-1 uses JFFS2. However, any efficient flash file system will work. Our contribution lies in the distribution of the file system over two connected devices.

D. Energy Management

Reducing the power consumption of mobile devices has been the subject of much research. Approaches include scaling the CPU voltage and frequency [9, 16, 23, 39], managing wireless interface usage [1], turning off banks of RAM [18, 21, 24, 27], or employing microsleep [20, 5]. In a larger device, such as a Stargate, these techniques still do not enable a power mode comparable to a mote device. Our architecture is designed to support devices that can operate at power levels separated by an order of magnitude.

Papathanasiou and Scott made an observation similar to ours: batching work, or increasing idle periods, leads to greater energy efficiency [28]. However, the goal of the their work was to increase burstiness in laptop disk drives.

The Wake-on-Wireless project (WoW) [35] 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. But, we have placed a large amount of functionality in the lowest tier. Our tier-0 system is capable of actually executing some tasks without waking tier-1.

E. Sensor Platforms

Recently, many 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 10-40mW and are tuned to

be highly power efficient. The primary design goal of these platforms is a small form factor, low cost, and very low power such that lifetimes of a few years can be obtained on two AA batteries.

There are also several more capable but still very power-efficient sensor nodes such as the Yale XYZ [25]. This node has dynamic frequency scaling capability and can operate between 2MHz and 56MHz with a power consumption of up to 3x greater than the mote. Such intermediate platforms can be used as clusterheads in applications that have moderate computation requirements. For instance while localization can be performed on such a node it cannot run a full database.

Our architecture targets resource-rich but power efficient sensor platforms that combine two processing elements—one small and one large. Two instances of such architectures are currently commercially available. The Stargate platform [6, 38] is the node that we use in our work. The PASTA node is an architecture that combines a trip-wire board with a DSP processor together with a PXA processor [34]. Other instances of such dual processor systems have been suggested in the literature although they are not commercially available.

VIII. Conclusion

This paper presents the design, implementation, and evaluation of Triage, a software architecture designed to minimize the energy consumption of a tiered hardware platform. It works by enabling a low-power tier to route tasks and batch them when possible. Our evaluation demonstrates that Triage yields a substantial energy savings for applications such as in-network storage and image processing.

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. K. Balan, J. P. Sousa, and M. Satyanarayanan. Tactics-based remote execution for mobile computing. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, San Francisco, CA, May 2003.

[3] S. Bandyopadhyay and E. J. Coyle. An energy efficient hierarchical clustering algorithm for wireless sensor networks. In *Proceedings of IEEE Infocom*, San Francisco, CA, March 2003.

[4] A. Basharat, N. Catbas, and M. Shah. A framework for intelligent sensor network with video camera for structural health monitoring of bridges. In *Proceedings of First International Workshop on Sensor Networks and Systems for Pervasive Computing (PerSeNS'05)*, Kauai Island, Hawaii, March 2005.

[5] 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.

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

[7] H. Dai, M. Neufeld, and R. Han. ELF: An efficient log-structured flash file system for micro sensor nodes. In *Proceedings of The Second ACM Conference on Embedded Networked Sensor Systems (SenSys '04)*, Baltimore, MD, November 2004.

[8] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–7, Santa Cruz, California, December 1994.

[9] 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.

[10] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002.

[11] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann. An evaluation of multi-resolution storage for sensor networks. In *Proceedings of The First ACM Conference on Embedded Networked Sensor Systems (SenSys '03)*, Los Angeles, CA, November 2003.

[12] D. Gay. Design of Matchbox, the simple filing system for motes. TinyOS Documentation, August 2003.

[13] 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.

[14] H. Wang L. Girod, N. Ramanathan, D. Estrin, and K. Yao. A platform for collaborative acoustic signal processing. Technical Report 46, UCLA/CENS, January 2005.

[15] 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 (SenSys '04)*, Baltimore, MD, November 2004.

[16] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the First ACM International Conference on Mobile Computing and Networking (MobiCom'95)*, Berkeley, CA, November 1995.

[17] Wendi Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocols for wireless microsensor networks. In *Proceedings of the Hawaiian International Conference on Systems Science*, January 2000.

[18] 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.

[19] A. D. Joseph and M. F. Kaashoek. Building reliable mobile-aware applications using the Rover toolkit. In *Proceedings of The Second ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, White Plains, NY, November 1996.

[20] 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.

[21] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Proceedings of the Ninth International Conference*

on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, November 2000.

[22] D. Li, K. Wong, Y. H. Hu, and A. Sayeed. Detection, classification and tracking of targets. In *IEEE Signal Processing Magazine*, March 2002.

[23] J. R. Lorch and A. J. Smith. Reducing processor power consumption by improving processor time management in a single-user operating system. In *Proceedings of the Second ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, Rye, NY, Novemeber 1996.

[24] V. De La Luz, M. Kandemir, and I. Kolcu. Automatic data migration for reducing energy consumption in multi-bank memory systems. In *Proceedings of the 39th conference on Design automation*, New Orleans, LA, June 2002.

[25] 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.

[26] 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.

[27] E. Musoll, T. Lang, and L. Cortadella. Exploiting the locality of memory references to reduce the address bus energy. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, Monterey, CA, August 1997.

[28] 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.

[29] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, Saint Malo, France, October 1997.

[30] S. Phatak and B. R. Badrinath. Data partitioning for disconnected client-server databases. In *Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access*, August 1999.

[31] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *Proceedings of the 1994 Summer USENIX Conference*, Boston, MA, June 1994.

[32] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[33] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[34] 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.

[35] 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.

[36] 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.

[37] T. Stathopoulos, L. Girod, J. Heidemann, and D. Estrin. System support for coordinated imaging for sensor networks. Unpublished Poster, 2004.

[38] 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.

[39] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of The First Symposium on Operating Systems Design and Implementation (OSDI'94)*, Monterey, CA, November 1994.

[40] D. Woodhouse. JFFS: The journalling flash file system. In *Ottawa Linux Symposium*, Ottawa, Canada, 2001.

[41] 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.