

# Energy-aware Mobile Service Overlays: Cooperative Dynamic Power Management in Distributed Mobile Systems

Balasubramanian Seshasayee, Ripal Nathuji, Karsten Schwan  
College of Computing  
Georgia Institute of Technology  
Atlanta, GA 30332  
{bala, rnathuji, schwan}@cc.gatech.edu \*

## Abstract

*With their increasingly powerful computational resources and high-speed wireless communications, future mobile systems will have the ability to run sophisticated applications on collections of cooperative end devices. Mobility, however, requires dynamic management of these platforms' distributed resources, and such management can also be used to meet application quality requirements and prolong application lifetimes, the latter by best using available energy resources. This paper presents energy-aware Mobile Service Overlays (MSOs), a set of mechanisms and associated policies for running mobile applications across multiple, cooperating machines while actively performing power management to extend system usability lifetimes. MSO policies manage energy consumption by (i) allocating application components to available nodes based upon their current energy capacities and resource availabilities, (ii) monitoring for, and responding to changes in energy and resource characteristics, and (iii) dynamically exploiting energy-performance tradeoffs in overprovisioned situations. Coupled with mobility, such cooperation enables multiple mobile platforms to bring their joint resources to bear on complex application tasks, providing significant benefits to application lifetimes and performance. Evaluations of MSOs on a MANET computing testbed indicate an extension in system lifetime of upto 10% for an example application.*

## 1 Introduction

Distributed applications running on Mobile Adhoc NETWORKS (MANETs) are being used in many domains, ranging from autonomous robotics, to online gaming, to emergency management. Growth in such applications is encouraged

by the proliferation of handhelds and portables and by substantial increases in the combined computational power of the MANET systems on which they run, the latter caused by the increased computational capabilities of these platforms and the improved connectivity afforded by their powerful and diverse communication devices. Also promoted by these trends are cooperative approaches to running applications across sets of participating machines, with early examples of these including robots that collaboratively undertake a search and rescue mission [11], distributed gaming, and ubiquitous computing environments.

Since cooperative mobile applications like those listed above operate in highly dynamic execution environments, their effective execution requires them to adapt, online, to changes in requirements and to dynamically react to changes in underlying platform resources, including reactions to disruptions caused by node mobility or failure [23]. Disruptions occurring in a team of cooperating robots, such as the failure of a robot that performs critical processing for the team, or the movement of this robot out of range, indicate that centralized solutions are not suitable for managing the application software components distributed amongst nodes. Realizing these facts, the research community has developed a wide variety of decentralized management services, including for energy management [17], load balancing [22], and QoS provisioning [29].

This paper focuses on energy management, where, for MANET applications, we develop *cooperative energy management strategies* that enhance energy profiles, (i) by decreasing energy consumption to the extent permitted by current application performance constraints, and (ii) by extending overall system and application lifetime, via migration of application services that are critical to the application away from energy constrained nodes. Specifically, concerning (i), for each single platform, we reduce its energy consumption by using common techniques for energy management, which include dynamic voltage and frequency scaling

---

\*This work is funded in part by the NSF-ITR award and Intel corporation

(DVFS). The resulting degradation in application execution can be reduced by utilizing memory-bound phases for such scaling [9], or, in real-time environments where there is a notion of slack, by increasing execution times (thereby reducing energy needs) only to the extent permitted by application deadlines [1]. Similar energy-performance trade-offs are available for other devices like memory, peripherals (network and disk interfaces), display, etc. Concerning (ii), we use computational offloading, whereby portions of the mobile workload are dynamically offloaded to nodes with better energy resources [22]. The former set of techniques have a direct effect on energy savings at distinct nodes, whereas the latter helps in longevity by sharing energy resources amongst multiple participants.

Energy-aware management is implemented for realistic mobile applications and systems using the lightweight and efficient Mobile Service Overlays (MSOs) [25]. MSO middleware provides efficient mechanisms for dynamically creating, moving, and reconfiguring computational services across distributed mobile platforms, and in addition, for monitoring underlying platform conditions. This paper uses these facilities to develop decentralized management protocols that (i) dynamically distribute and re-distribute application components among participating nodes, considering the overlay routes that satisfy the application’s latency requirements while at the same time, determining the most energy-efficient allocations, (ii) recover unused portions of resources in an overprovisioned system with little or no impact on application performance, and (iii) use de-centralized online monitoring and reconfiguration to locally, and thereby, with low delay and overhead, respond to dynamic changes in application requirements and environment conditions. The management algorithms being used, specifically the algorithm for dynamic resource reclamation, are experimentally demonstrated to track optimality, with low overhead. MSOs using this algorithm – *energy-aware MSOs* – offer notable benefits. On a wireless, multi-hop ad-hoc network of handheld computing platforms, for instance, an energy-aware MSO extends system lifetime up to 10% for a five-node network.

The remainder of this paper is organized as follows: In Section 2, we explore related research, then describe an overview of energy-aware MSOs in Section 3. This is followed by a description of the energy management techniques used in MSO in Section 4. Section 5 discusses current trends in power management. We present selected elements of the implementation of energy-aware MSOs and evaluate it experimentally in Section 6, followed by conclusions and future work in Section 7.

## 2 Related Work

Prior research on energy conservation in ad-hoc networks has mainly focused on energy-aware routing proto-

cols – by improving existing protocols like AODV [21], by factoring the energy levels of each node in the routing cost metric [8], or through novel protocols like probabilistic routing [26]. The former classifies nodes based on their energy levels, and uses this instead of hop count to determine an energy-optimal route. The latter uses a similar cost metric by aggregating the energy values of the nodes in each route, then randomly chooses a route with a probability proportional to the cost metric along the route. While these approaches are suitable for network-bound applications and sensor nodes, where the network interface accounts for a significant portion of the power budget, for the applications and platforms considered in our work, experimental results indicate that the energy consumed by the network interface is quite small compared to CPU energy. Our research, therefore, primarily leverages prior work on reducing CPU energy consumption, including reducing energy usage by applying dynamic voltage and frequency scaling [10] on multiprocessor systems. The benefits of DVFS for dynamic slack reclamation in real-time multiprocessor systems has been evaluated in prior research [15]. A static schedule is first constructed for periodic tasks, then slack reclaiming is used to save power, while satisfying the real-time constraints. Resource reclaiming in multiprocessor real-time systems has been dealt with in great detail in [27], where the authors describe two algorithms to perform online reclaiming on a static schedule.

Computational offloading has been used extensively for power-aware load balancing, for environments ranging from clusters of workstations [22] to embedded devices [28]. The latter performs computational offloading in conjunction with setting CPU frequencies, to minimize energy consumption. Distributed middleware can be useful in managing computational entities. A survey of various middleware implementations for mobile environments has been performed in [16]. Relevant projects at Georgia Tech include [3] and DFuse [13], the former implementing runtime methods for adjusting overlay configuration to end points’ mobility behavior. Research efforts to include mobile nodes in Grid technology include [2], which proposes a mobile agent framework to provide/use Grid services at the mobile nodes, so that distributed resources from the Grid can be accessed by such users. MagnetOS [14] uses a distributed framework to partition a monolithic Java application into its constituent classes for cooperative execution on a MANET. Efficient placement of application components is carried out by monitoring the data traffic among nodes, and using distributed algorithms to shorten the mean path length of data.

## 3 Mobile Service Overlays – System Overview

*Mobile Service Overlays* (MSO) [25] is a distributed

middleware system designed to execute mobile and pervasive applications. MSO provides mechanisms to dynamically create and manage overlays on a mobile network. It operates under an event-driven computation model [16], where data, in the form of events, is exchanged across distributed computing nodes for processing. MSO models applications as directed flow graphs whose vertices represent processing performed on events arriving via directed edges. Associated with each edge is a format describing the data passing through the edge. Such a graph-based representation implicitly captures the data dependencies among computational entities, and it also simplifies application partitioning for a distributed environment. MSO programs, therefore, consist of code modules running in vertices mapped to overlay nodes, receiving formatted inputs from and generating outputs to other vertices. Similar to an event processing system, the flow graph consists of event sources and sinks.

MSO targets dynamic management of the application flowgraph in unreliable, distributed mobile environments. Correspondingly, its design goals include (1) decentralized resource management and (2) low overhead mechanisms for (re-)deployment. No explicit organization is enforced among the participating nodes—all MSO nodes are peers and have identical middleware functionality. Furthermore, to avoid the scalability and reliability issues arising from centralized or global management, the application flow graph is partitioned into its constituent and independently manageable computational *chains*. Formally, a chain is a maximal set of sequential vertices and edges of the flow graph, with a single entry and a single exit. Events enter the first vertex of the chain, the *head*, sequentially pass through and are operated on at each node in the chain, and finally exit at the last node, the *tail*. Many application properties (e.g., end-to-end delay) can be formulated to depend on the corresponding local properties of each chain, and different chains can be managed independently, to guarantee such properties. In this fashion, chains compartmentalize management to be confined to more “local” portions of the application. Figure 1 shows an example flowgraph decomposed into its constituent chains ( $A \rightarrow D$ ,  $D \rightarrow E$ ,  $F \rightarrow G$ ,  $G \rightarrow D$ ,  $G \rightarrow I$ ). Using the partitioning provided by the chain abstraction, MSO provides multiple low level services to higher layers for use in cooperative MANETs:

**Deployment** consists of instantiating the application flowgraph onto the mobile network. This is achieved by first assigning each vertex to a node in the network. The route from the source to sink node is determined, and vertices of a chain are assigned to nodes in the route in proportion to their capacities. Next, the overlay network is constructed by creating the vertices at the assigned nodes and the links connecting them. The chain abstraction allows these steps to be carried out in parallel, as each chain head carries out

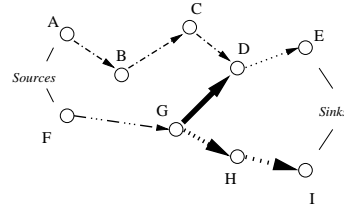


Figure 1. Partitioning the graph into chains

vertex assignment and overlay network construction locally for that chain. The deployment algorithm is discussed in detail in [25].

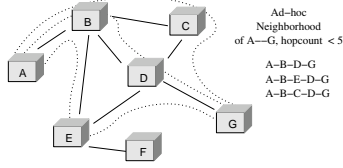
**Monitoring** of metrics like CPU utilization, data transfer, expected battery lifetime and liveness of neighbors are available for use by the higher layers for reconfiguration triggers and decisions. These are also made available to other nodes, through RPC calls.

**Reconfiguration** involves remapping portions of the overlay network to create a different assignment between vertices in the overlay and underlying machines. MSO provides capabilities to perform remapping at various granularities: (i) Intra-chain remapping, where a single vertex is *relocated* from one node to another, finds use in responding to local changes, (ii) Chain remapping, where existing assignments of vertices in a chain are freed and a fresh assignment performed, and (iii) Multiple chain remapping which extends chain remapping to many chains. This is useful in responding to more global changes, but used rarely due to its higher cost.

**Self-Management** is a generic framework in MSO that involves (i) monitoring for specific changes in observed metrics, and (ii) triggering reconfiguration mechanisms at the appropriate granularity to counter the change, based on a predefined ruleset. As MSO is decentralized, any node that observes a change can trigger a reconfiguration according to predefined rules tailored towards specific objectives. Further, as this is a generic procedure, it can be applied to different management goals like load balancing, mobility & fault tolerance, latency minimization, etc. In this work, we use a simple greedy ruleset, to perform reconfiguration to minimize energy consumption.

## 4 Techniques for Energy Management

Energy management in a dynamic environment is a continuous process, requiring an energy-aware assignment, followed by online monitoring to trigger actions that shift the system towards optimality. MSO provides three techniques for energy management, viz., energy-aware allocation, reallocation, and dynamic resource reclaiming.



**Figure 2. Ad-hoc Route Neighborhood**

#### 4.1 Energy Aware Allocation

The chain deployment procedure (detailed in [24]) is modified to address energy management concerns, using the following techniques: (i) modifying route exploration to include *Ad-hoc Route Neighborhoods* and (ii) using the *Global Lifetime Sustainability* heuristic to determine the best assignment of vertices to nodes from among the routes in the neighborhood.

**Ad-Hoc Route Neighborhoods:** The allocation algorithm in MSO uses the route explored from the source node(s) to the sink(s) to perform the assignment over each chain. However, the route thus found is entirely dependent on the ad-hoc routing protocol employed in the underlying layer. While protocols like AODV, DSR, etc. typically use the shortest route, recent research has explored a variety of techniques for power management at the network routing layer, including probabilistic routing [26], multipath routing, and using multiple radios. Hence, prior to allocation, no assumptions can be made by MSO about the underlying protocol’s behavior. As a result, MSO explores all possible routes that can be taken from the source to the destination (bounded by a maximum hopcount), and chooses the “best” route from among those for the assignment. The set of all the routes discovered in this manner is termed the *ad-hoc route neighborhood* of the chain. The end-to-end latency requirements of the chain limits the maximum hop count for the route, and consequently, the length of each route in the ad-hoc neighborhood, and the number of routes obtained. Figure 2 illustrates this concept, enumerating the ad-hoc route neighborhood of routes from node A to G in the topology, with hopcount strictly less than 5.

We rely on a distributed protocol to find the route neighborhood of a chain, given constraints that include maximum latency and maximum hopcount. The protocol is similar to that used by AODV for route discovery [21], with two differences: (i) the destination node stores all the routes it obtains, and (ii) after the maximum latency period, the source node queries the destination to directly obtain all the routes stored. This protocol enumerates all of the routes satisfying the latency constraints. Since no state is saved in any nodes (except at the destination), its overhead is low. Further, each packet not reaching the destination is eventually dropped, as the monotonically increasing hopcount reaches the upper bound.

**Global Lifetime Sustainability (GLS) Heuristic:** The placement of software components during allocation and reallocation can directly affect the lifetime of a MANET scenario. In the simplest case, with two nodes and a single component, energy can be depleted fairly and system lifetime is maximized by migrating the workload to the participant with larger battery capacity whenever a reallocation is triggered. When there are multiple components and multiple nodes, all with varying requirements and energy levels, optimal decisions cannot be made within reasonable complexity constraints. Instead, we use a heuristic to determine where to place various components. In particular, we define a GLS metric for a set of candidate nodes and energy levels and utilize a heuristic that attempts to maximize this metric and/or minimize potential decreases in metric values. Specifically, for this work, we define the GLS metric to be the product of the remaining energy levels on nodes under consideration when deciding where to place a software component. This approach works well for the scenarios addressed here since nodes are treated equally and are homogeneous. We note that the GLS metric can also be effective when used in more general settings, to express variations among nodes in heterogeneous environments (since power consumption profiles vary accordingly) and when tuning allocation policies at runtime. We will investigate these issues in future work.

Chain assignment decisions in MSO use the GLS heuristic and ad-hoc neighborhoods. Intuitively, from among all possible routes discovered (ad-hoc route neighborhood) for each chain, (1) a greedy assignment is determined based on resource availability and the GLS heuristic, and (2) a cost is associated with the route and allocation scheme. The route with the smallest cost is then chosen for the chain. In particular, we define the cost as the projected difference in the GLS metric based upon the allocation over some period of time. Additional detail about the algorithm appears in [24].

#### 4.2 Energy-Aware Reallocation

Energy-Aware reallocation consists of moving application components in response to changing conditions. It utilizes the monitoring, management, and reconfiguration services provided by MSO (see Algorithm 1). Here, local variations in node lifetimes are overcome by relocating vertices to neighboring nodes with higher lifetimes. Global variations are addressed by remapping entire chains, but this is performed less frequently due to its higher costs.

#### 4.3 Workload-Aware Dynamic Resource Reclaiming

To conserve energy in overprovisioned nodes, we design a distributed protocol that explores energy-performance tradeoffs in a distributed system through resource reclaiming – i.e., recovering any resource from the system to the extent that it does not affect the performance and hence, the quality of the application. Such resources include periph-

---

**Algorithm 1** Energy-Aware Reallocation

---

- 1: Each node  $n$ , periodically queries the expected lifetime of all of its neighbors
  - 2: Select node  $n'$  s.t.  $lifetime(n')$  is the largest among all neighbors
  - 3: **if**  $lifetime(n') - lifetime(n) > threshold$  **then**
  - 4:   **while**  $\exists v'$  in  $n$  not considered for relocation **do**
  - 5:     Select vertex  $v$  s.t.  $Cost(v) > Cost(v')$ ,  $\forall v'$  in  $n$  not considered for relocation
  - 6:     **if**  $v$ 's relocation to  $n'$  does not affect event latencies along the chain **then**
  - 7:       Relocate  $v$  to  $n'$
  - 8:       Break out of while loop
  - 9:     **end if**
  - 10:   **end while**
  - 11: **end if**
  - 12: During long periods of idleness,  $\forall v$  housed at  $n : v$  is a chain head, check for any changes in the ad-hoc neighborhood, and remap the chain if a more optimal neighborhood is found.
- 

eral interfaces like storage (via sleep modes), memory (via switching off banks), and CPU (dynamic voltage/frequency scaling). In this paper, we demonstrate this approach with CPU-based techniques.

Each node attempts to reclaim as many of the resources as possible to minimize energy consumption, then distributes the remaining opportunities to other nodes. Event sources in MSO associate a deadline with each event, sending it along with the event itself. Where this is not available/known, the event inter-arrival period is used. When the processed event reaches the sink, its slack is computed, i.e., the time difference between the deadline of the event processing and the actual time of completion. A positive slack value serves as a measure of overprovisioning, in that unnecessary effort was expended in processing the event. This presents an opportunity to ‘reclaim’ the slack by scaling down the voltage/frequency in some/all of the CPUs involved in event processing, thereby reducing energy consumption. An implicit assumption made here is that the nodes’ clocks are synchronized.

Slack reclaiming starts at the sink node, since only it can compute the slack value. Starting from this node, each node, on obtaining the slack values from all its downstream nodes (a node  $n_1$  is downstream to  $n_2$  if  $\exists v_1, v_2 : v_1$  is assigned to  $n_1, v_2$  to  $n_2$ , and  $\exists$  a directed edge from  $v_2$  to  $v_1$ ), attempts to scale down its own CPU frequency/voltage so as to maximize energy savings. Next, it computes the slack available to each of its upstream nodes and sends them these values. Starting at the sink node(s), the algorithm thus propagates towards the source(s).

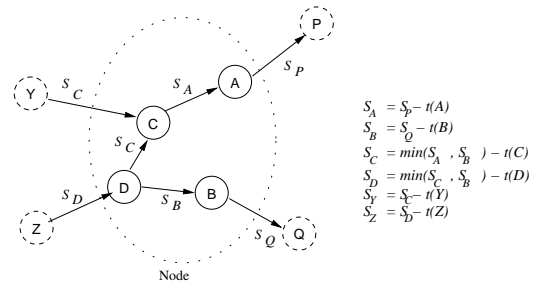
Within each node, Algorithm 2 is used to determine its

---

**Algorithm 2** Slack reclamation

---

- 1: **repeat**
  - 2:   **if**  $\exists v : \forall v'$  and  $v \rightarrow v'$ ,  $v$  has received  $slack(v')$ , **then**
  - 3:     **for all**  $f : f$  is a CPU frequency **do**
  - 4:       Set  $slack_f(v) = \min_{v'} \{slack_f(v')\} - t_f(v)$ ,  
      { where  $t_f(v)$  is the estimated execution time of the computation of  $v$  at frequency  $f$  }
  - 5:        $\forall v_p : v_p \rightarrow v$  and  $v_p$  is in the same node as  $v$ , send  $slack(v)$  to  $v_p$
  - 6:     **end for**
  - 7:   **end if**
  - 8: **until** all the vertices in the node have been considered  
      {Slacks for each frequency have been computed}
  - 9: **for all**  $v : v'' \rightarrow v$ , and  $v''$  is not in this node, **do**
  - 10:     $f_{choose}(v) \leftarrow \min\{f\} \text{ s.t. } slack_f(v) \geq 0$
  - 11: **end for** { The minimum feasible freq. for each vertex }
  - 12: Set the new frequency,  $f_{new} = \max\{f_{choose}(v)\}$
  - 13:  $\forall v : v'' \rightarrow v$ , and  $v''$  is not in this node, send  $slack_{f_{new}}(v)$  to  $v''$
- 



**Figure 3. Slack propagation within a node**

best CPU frequency. It computes the minimum feasible frequencies for each vertex (lines 1-11) in a node, then chooses the maximum from among them as the frequency for that node (line 12), finally propagating this value to all vertices upstream (line 13). Figure 3 illustrates this procedure, with the slacks along the edges represented by  $S_x$ 's and the execution time of the computation at each vertex represented by  $t_x$ 's, and shows how the slack is ‘consumed’ by each node and the remainder passed on to vertices upstream.

Being a greedy algorithm, its focus is to quickly reclaim any slack made available, hence it seeks only local optima. As a result, the closer a node is to the sink, the greater will be its energy savings. Fairer methods for slack reclamation would require additional coordination among MSO nodes, thereby introducing additional protocol complexity. The main assumption in this algorithm is that the expected execution time at each frequency for the vertices can be estimated. Research efforts in workload characterization can be used to perform such estimates [4]. Even with accurate ap-

plication characterization, factors like network jitter, mobility related effects, or even changing application needs can cause changes in slack availability. The response time to such events primarily depends on the frequency at which slack reclamation is initiated. Additionally, since a side effect of the greedy algorithm results in concentrating the bulk of reclaimed slack closer to the sink nodes, this also enables quick reversion of the reclaimed slack, during such conditions.

## 5 Power Tradeoffs and System Implications

In order to effectively manipulate energy usage amongst distributed nodes, it is essential for MSO to understand the power and performance tradeoffs of the underlying hardware. In this section, we present results from detailed power measurements of our evaluation platform which provide the intuition and tradeoffs that drive the system’s power management policies.

The hardware environment used in our experiments is the Intel Sitsang platform, with a PXA255 processor, and 64MB RAM. It runs the Linux-2.4.19 kernel, modified with Xscale and platform specific patches. Each node also has an 802.11b wireless interface, in ad-hoc mode, in addition to a 10Mbps base-T ethernet interface. All power measurements are performed using a Tektronix TDS5104B oscilloscope, Tektronix TCP202 current probes, and Tektronix P6139A voltage probes.

### 5.1 Platform Power Trends and DVFS

The Sitsang platform is designed around a PXA255 XScale processor. The processor supports frequency and voltage scaling via multiple operating points that vary CPU frequency as well as the frequency to the internal PXA bus, thereby affecting latency to memory and I/O devices. The core frequencies available are 400MHz, 300MHz, 200MHz, 150MHz, and 100MHz. Though multiple bus frequencies are plausible for certain core frequencies, for the purposes of analysis and experiments in this paper, we always utilize the maximum bus speed possible for a given core speed.

For power analysis, we utilize a tunable synthetic workload that has characteristics similar to the robotics applications used in our MSO research [25]. Specifically, for these applications, we find memory access behavior to be an attribute that can vary significantly. Software components like Bayesian classifiers are CPU bound, where performance scales with frequency, whereas image analysis like blob finding displays increased memory activity due to footprints larger than the 32K cache size on the PXA255 processor. In order to provide a fair comparison across this attribute, we have developed a synthetic workload that can be tuned to vary memory boundedness while maintaining the amount of work (i.e. instruction count) performed. This benchmark is used in subsequent evaluations.

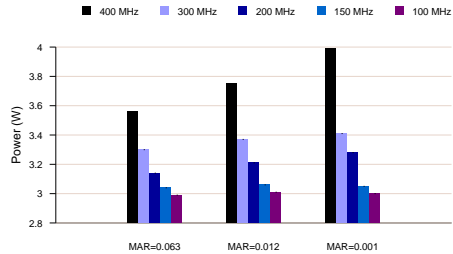


Figure 4. Sitsang Active Platform Power Consumption

When completely idle, the Sitsang consumes 2.5W-2.64W depending upon the operating point to which it is set. A more significant variation can be observed between the different frequencies when active, as illustrated in Figure 4. The figure provides power data when the platform is one hundred percent utilized and executing workloads with varying memory accesses per instruction (MAR), a parameter that in turn affects the cycles per instruction (CPI) required to execute each application. As expected, we see decreasing power consumption as frequency is reduced, with the difference between 400MHz and 100MHz being as high as 25%. These system-level trends underscore the possibilities of energy savings available via DVFS. We also observe from the figure that the system power is not only a function of frequency, but can also vary significantly based upon workload characteristics. Indeed, at 400MHz, the power varies by as much as 11% between the different applications. This highlights the potential benefits and necessity of online monitoring in MSO to dynamically tune energy management for application specific behavior.

From the results in Figure 4, it is clear that reducing the operating point of our platform, when possible from a performance standpoint, can reduce power consumption. The resulting energy savings, however, are not quite as apparent. For periodic applications, frequency can be reduced when there is slack without a performance penalty. This reduction increases the active portion of a period while reducing the idle period. As recent work has shown, reducing processor frequency may result in reduced power consumption during active portions of the period, but it can also increase energy consumption after some point of slack reclaiming [18, 10]. The existence of these counterintuitive trends can be affected further by the presence of other power management schemes with which DVFS must coexist [7, 19]. We consider these trends by obtaining the *cycle energy*, the combination of the active and idle energy signatures in a period, of the three MAR varying applications across different CPU utilizations in Figure 5.

Figure 5(a) illustrates the tradeoffs of the different op-

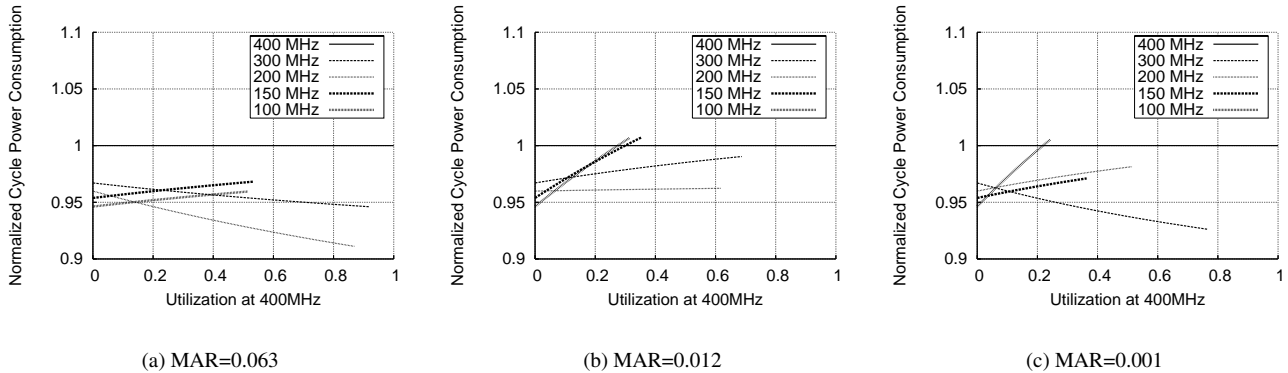


Figure 5. Platform Cycle Energy vs. CPU Utilization

erating points across utilization behavior for a memory-bound workload with resultingly high CPI. As utilization increases, it becomes infeasible to execute at certain frequencies until eventually, only the highest operating point can maintain the performance of the application. Since the application is memory-bound, the performance of reduced frequencies can closely match those of higher frequencies, especially when the bus frequency can be maintained. This is exhibited between the 300MHz and 200MHz operating points as well as the 150MHz and 100MHz frequencies by the fact that the respective energy curves end at about the same utilization (i.e., the modes become unusable at similar load). The reason for this is that the performance of the application is driven by bus frequency instead of core frequency due to the high memory access rate. We can see from the figure that the optimal operating point is only the smallest one possible for very low utilizations, after which 200MHz is optimal even though 150MHz and 100MHz would be options as well. Similar inflection points can also be observed for lower MARs in Figures 5(b) and 5(c), though in the latter the optimal operating point gets pushed even further to 300MHz at high utilizations. *These trends show that the energy optimal operating point can vary based upon workload characteristics as well as the utilization required to execute the workload.* It should be noted that even simply monitoring MAR is not adequate (our own results show data stalls per cycle should also be monitored), a full list of required attributes can be obtained via existing workload characterization research [4].

The platform power trends discovered in our experiments directly affect DVFS-based energy management in MSOs. First, they show that when utilizing DVFS to dynamically tune energy behavior via slack reclaiming of periodic applications, MSO middleware must monitor resource utilization information for software components so that it can be aware of ‘where’ the system is located along the utilization curve, thereby determining the minimum oper-

ating point to utilize at a particular node. Second, with on-line monitoring, MSO can also determine the performance scalability of an application by determining metrics such as MAR and the resulting data stalls per cycle. By coupling this information with platform power characteristics, MSO can more readily determine application specific inflection points at runtime than can be done by static policies.

## 5.2 Wireless Communication Overheads

In addition to platform energy savings with respect to utilizing frequency scaling, our approach also exploits cooperative systems by offloading computations to take advantage of remote resources and energy reserves. This type of offloading has been shown to provide significant system lifetime benefits in our previous work [20]. Here, we continue to leverage this type of energy management by considering the possibility of migrating software components in MSOs during reallocations. A question that arises, however, is how the associated communication energy overheads compare to the benefits of offloading. To obtain insights into this trade-off, we stream data between two Sitsang platforms over a wireless link. We then monitor the system, CPU, and radio power of one of the systems at different data rates of UDP/IP. These experiments result in the following findings. First, we find that the link becomes saturated at 4Mbps. At this extreme, system power consumption increases by 300mW, while the radio power signature is only elevated by 90mW. The CPU power signature explains this discrepancy, as we observe that the processor is consuming active power during 25% of the time due to packet processing overheads. Therefore, the majority of the power increase can be captured by simply monitoring system utilization. The reason for the minimal increase in radio power consumption, even at high link utilization, is that in ad-hoc mode, the radio cannot be placed into a sleep state—it is always in a promiscuous read mode, the power signature of which varies little from sending. Since the radio power consumption changes

negligibly with use, the overheads of utilizing it, for the sake of our flowgraphs with little communication utilization, can be effectively ignored. Therefore, in MSO energy management, we only consider the computational overheads of software components when performing energy load balancing.

## 6 Experimental Evaluation

The approaches described in this paper are implemented in the MSO middleware. This middleware is written entirely in C/C++, and uses an overlay construction toolkit, termed EVPath [6]. EVPath allows the creation of entities called ‘stones’ that operate on events. Stones can be linked via a variety of network transport protocols. EVPath also provides a SOAP interface using gSOAP [5] to enable remote management of stones. AODV [12] is used for ad-hoc routing, but MSO is independent of the underlying routing protocol. Experiments are performed with the Sitsang handheld devices described earlier. Power consumption at each node is estimated through a daemon that periodically monitors the CPU usage and computes energy consumed based on the CPU frequency and with an application dependent power model (which in turn is obtained by power measurements performed using the oscilloscopes on an instrumented node).

### 6.1 Reallocation

The first set of experiments uses two Sitsang nodes running a single application (CPU intensive, with 30% utilization), such that only one node runs at any time, with the other node is idle. By monitoring each others’ battery levels, the nodes can cooperatively run the application so that they maximize their battery lifetimes. Overall lifetime of the system is defined as the lifetime of the first node that exhausts its battery completely. Starting from 2kJ for each node, the polling frequency for monitoring is increased, and the corresponding increase in node lifetimes are observed. (Figure 6). The results are compared to a static assignment, where one node is always idle and the other is always busy. As we increase the frequency of reallocation, the system lifetime increases, but more time was also devoted in performing the reallocations than performing the actual work. The net yield thus shows an increase, followed by a decrease, with the highest increase found to be roughly 11.5%

Next, we study an example on a five node testbed, with the topology in Figure 7(a). The application flow graph is chosen to accommodate all three different kinds of event flow combinations possible, i.e., linear, split, and join flows. Each node runs an instance of MSO, and begins with a fixed energy level. The threshold for reallocation decisions is set to 50J, and the frequency of polling (for monitoring neighbors’ energy levels) is chosen to be 25 sec. All application components are CPU bound, and while two of them run at 80% CPU utilization, the other three run at 15%.

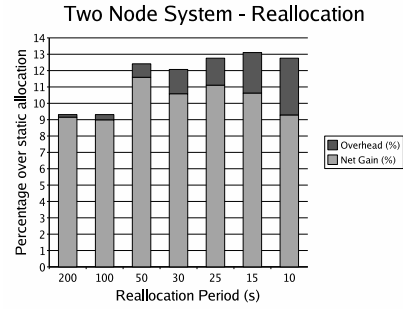


Figure 6. Reallocation in Two Nodes

Such a workload is representative of applications that perform different pipelined processing on data. For instance, in an image processing pipeline in the robotics domain, the pre-processing stages like edge detection, scaling, etc. are data intensive, whereas applying learning algorithms on the images are typically CPU intensive. Events of size 4KB are sent via a single source at the rate of 1 per 1.5 seconds. For this experiment, we ignore the timeliness of event delivery (causing the check in line 6, in algorithm 1 to always return true). This enables aggressive relocation of vertices to the best available neighbors. We discuss the benefits of using MSO along two parameters: (i) the system lifetime of a cooperating group and (ii) the parity in the lifetimes of the nodes forming the group. An example of the first type of requirement is in collaborative tasks that require the participation of all mobile nodes. The second rule can be useful in enforcing a uniform policy for all participants in the task. For our purposes, we term the lifetime of the first dying entity in the group as the system lifetime, and we quantify lifetime parity by measuring the standard deviation of individual node lifetimes. We observe the trend of these metrics, as the initial energy available with the nodes are varied. We compare the results of our reallocation with a static assignment (Figure 7). The difference in the lifetimes afforded by these strategies increases, as the initial energy increases, with the differences being close to 10% at 5kJ. This is a consequence of the fact that the lifetime of the node(s) executing the computationally intensive components of the application flow graph exhibits a linear relationship with the initial energy. However, as reallocation shifts the heavy computation among all the nodes, this effect is mitigated. Similarly, the lifetime parity with reallocation shows no particular trend with increasing initial energy, as against the widening gap in node lifetimes observed with a static allocation.

### 6.2 Dynamic Resource Reclaiming

The next set of experiments evaluate the dynamic resource reclaiming algorithm, over a synthetic workload. As discussed previously, we apply this technique to source-

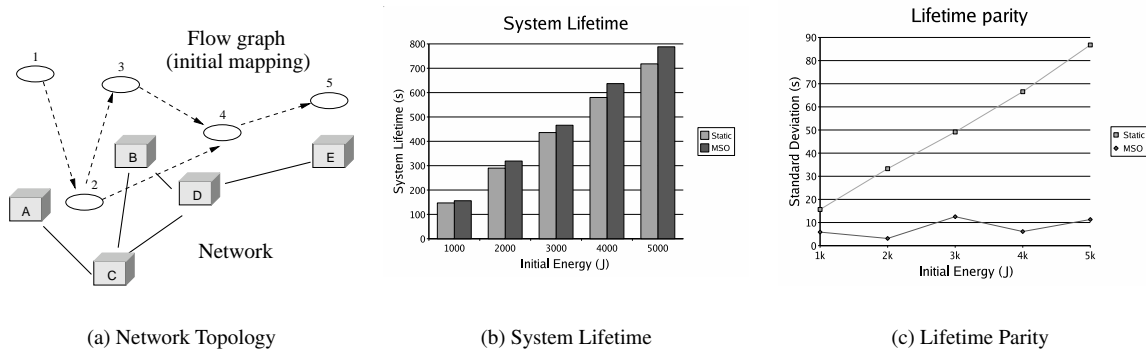


Figure 7. Reallocation

defined event slack available at the application sink node. The setup for the experiment consists of five Sitsang nodes in the same configuration as in the previous study. Each vertex executes a memory-intensive synthetic application. The application is run at three different scenarios, corresponding to having a CPU utilization of 40%, 60% and 80%, by increasing the duration of execution of each event in the application. Events of size 8 KB are sent with a periodicity of 2.5 seconds. The execution time of each event, when run at 80% utilization and the highest frequency (400 MHz) is found to be roughly 2s at each node.

The resulting energy consumption of the application is measured in the presence of the resource reclaiming algorithm. This figure is compared against a choice of frequencies for each possible slack value that is optimal in that it minimizes energy consumption. The energy values are normalized against the default case where all nodes run at the highest frequency. As shown in Figure 8, the algorithm is found to closely track the most optimal settings. In some cases, especially at the top frequencies, our algorithm appears to outperform even the optimal solution, but this is only because of missed deadlines, i.e., the algorithm reclaims more slack than available, thus saving more energy but hurting performance. This is due to errors in predicting the execution time at various frequencies, and other sources of error in event delivery. For the higher utilization workload, as both the slack available, as well as the execution time are high, any errors in estimation can cause large deviations from the optimum frequency settings.

Finally, we evaluate the overheads of various strategies that can be employed for resource reclaiming. We consider three strategies in this study: (i) Aggressive, where slack is polled frequently (every 30 sec), and positive as well as negative slacks are immediately propagated throughout the network, (ii) Conservative, which polls for slack less frequently (60 sec), and (iii) Opportunistic, where positive slacks are propagated at a low rate (60s), and negative slacks

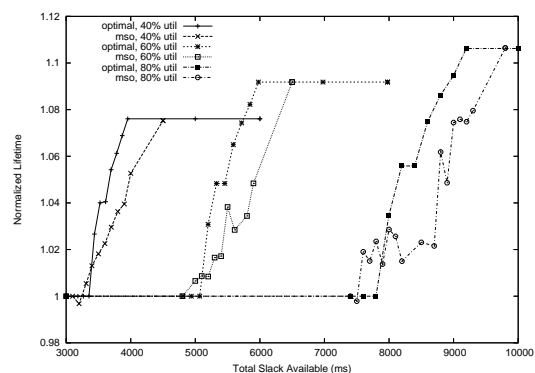


Figure 8. System energy savings with slack reclaiming

Table 1. Slack Reclaiming – Strategies

Strategy	Mean slack (ms)	Mean  slack  (ms)	Reclaims
None	1188	1204	0
Aggressive	-23	305	1012
Conservative	59	245	528
Opportunistic	-45	215	822

at a high rate (30s), so as to conserve energy without a high overhead, but react relatively quickly when performance is hurt. The event traffic was such that all the ranges of slack values were used in the test, over a period of about 15 minutes. The results of these strategies are shown in Table 1.

The aggressive strategy is more prone to mispredictions and overcorrections, than the others, but it is able to utilize all of the slack. The conservative approach is relatively more stable, with almost half the number of reclaims, but it can let the slack available for shorter periods of time go unrecovered. The opportunistic strategy strikes a balance between these extremes, to react quickly during performance critical phases alone. Correspondingly, the number of slack

reclaims also lies between these strategies.

## 7 Conclusions and Future Work

This paper discusses the problem of cooperative energy management in distributed mobile systems. It presents decentralized management techniques when running a distributed application in a MANET environment. These include (i) energy aware allocation, followed by (ii) reallocation in response to changing energy conditions, and (iii) dynamic resource reclaiming in overprovisioned systems. The algorithms are evaluated on a MANET testbed of handheld computing devices, indicating the feasibility of our approach. Further, our experiments indicate that the energy consumption of the network interface in handheld devices like the Sitsang platform form a tiny portion of the overall power consumption, underscoring the need to focus on other power hungry components of similar systems.

Our future work involves extending MSO to heterogeneous networks with asymmetric nodes, by including mobile phones, laptops and other portable computing devices. We are also currently evaluating MSO with prototype robot devices, with the aim of studying the tradeoffs between policy-enabled quality of service and energy consumption for typical robotic applications.

## References

- [1] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. In *IEEE Transactions on Computers*, 53(5), May 2004.
- [2] D. Bruneo, M. Scarpa, A. Zaia, and A. Puliafito. Communication paradigms for mobile grid users. In *CCGrid*, 2003.
- [3] Y. Chen and K. Schwan. Opportunistic overlays: Efficient content delivery in mobile ad hoc networks. In *Middleware*, 2005.
- [4] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2004.
- [5] R. Engelen. Code generation techniques for developing light-weight xml web services for embedded devices. In *ACM SAC*, 2004.
- [6] <http://www.cc.gatech.edu/systems/projects/EVPath/>.
- [7] X. Fan, C. Ellis, and A. Lebeck. The synergy between power-aware memory systems and processor voltage scaling. In *Proceedings of the Workshop on Power-Aware Computer Systems (PACS)*, December 2003.
- [8] N. Gupta and S. Das. Energy-aware on-demand routing for mobile ad hoc networks. In *Workshop on Distr. Comp., Mobile and Wireless Comp.*, 2002.
- [9] C. Isci and M. Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2006.
- [10] R. Jejurikar and R. Gupta. Dynamic voltage scaling for system-wide energy minimization in real-time embedded systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, August 2004.
- [11] J. Jennings, G. Whelan, and W. Evans. Cooperative search and rescue with a team of mobile robots. In *ICAR*, pages 193–200, July 1997.
- [12] <http://w3.antd.nist.gov/wctg/aodv-kernel/>.
- [13] R. Kumar, M. Wolenez, B. Agarwalla, J. Shin, P. Hutto, A. Paul, and U. Ramachandran. Dfuse: A framework for distributed data fusion. In *ACM Sensys*, 2003.
- [14] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *Mobisys*, 2005.
- [15] J. Luo and N. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *ICCAD*, 2000.
- [16] C. Mascolo, L. Capra, and W. Emmerich. Mobile computing middleware. *Advanced Lectures on Networking, LNCS*, 2002.
- [17] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. Giuli, and X. Gu. Towards a distributed platform for resource-constrained devices. In *ICDCS*, 2002.
- [18] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing*, June 2002.
- [19] R. Nathuji, K. O'Hara, K. Schwan, and T. Balch. Compatpm: Enabling energy efficient multimedia workloads for distributed mobile platforms. In *Proceedings of the ACM Multimedia Computing and Networking Conference (MMCN)*, 2007.
- [20] K. O'Hara, R. Nathuji, H. Raj, K. Schwan, and T. Balch. Autopower: Toward energy-aware software systems for distributed mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2006.
- [21] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, 1999.
- [22] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath. Dynamic cluster reconfiguration for power and performance. *Compilers and Operating systems for low power*, 2003.
- [23] C. Prehofer and C. Bettstetter. Self-organization in communication networks: principles and design paradigms. *IEEE Communications Magazine*, 2005.
- [24] B. Seshasayee, R. Nathuji, and K. Schwan. Energy-aware mobile service overlays: Cooperative dynamic power management in distributed mobile systems. Technical report, GIT-CERCS-07-05, Georgia Tech, 2007.
- [25] B. Seshasayee and K. Schwan. Mobile service overlays: Reconfigurable middleware for manets. In *MobiShare*, 2006.
- [26] R. Shah and J. Rabaey. Energy-aware routing for low energy ad hoc sensor networks. In *WCNC*, 2002.
- [27] C. Shen, K. Ramamritham, and J. Stankovic. Resource reclaiming in multiprocessor real-time systems. In *IEEE Transactions on Parallel and Distributed Systems*, 1993.
- [28] R. Xu, D. Zhu, C. Rusu, R. Melhem, and D. Mosse. Energy efficient policies for embedded clusters. In *LCTES*, 2005.
- [29] Q. Xue and A. Ganz. Ad hoc qos on-demand routing in mobile ad hoc networks. *Journal of Parallel and Distributed Computing*, 63, 2003.