DESIGNING DISTRIBUTED SYSTEMS FOR INTERMITTENT POWER

A Dissertation Presented
by
NAVIN KUMAR SHARMA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

May 2013
Computer Science
DESIGNING DISTRIBUTED SYSTEMS FOR INTERMITTENT POWER

A Dissertation Presented
by
NAVIN KUMAR SHARMA

Approved as to style and content by:

______________________________
Prashant Shenoy, Chair

______________________________
Deepak Ganesan, Member

______________________________
James Kurose, Member

______________________________
Michael Zink, Member

______________________________
Lori A. Clarke, Department Chair
Computer Science
To my parents and my brothers, Santosh and Mukesh.
ACKNOWLEDGMENTS

I would first like this opportunity to thank my PhD advisor Prof. Prashant Shenoy without whose guidance I would not have received a doctorate. Over the years he has helped me transition from a system engineer to a system researcher who combines strong development skills with research skills to build real systems that impact real people. It was due to his guidance and mentoring skills that I have enjoyed my academic career more than my professional career.

Next, I am deeply indebted to Prof. David Irwin who has been my (unofficial) co-advisor for the last five years. His guidance and invaluable comments have led me to finish my PhD work in a timely fashion. I would also like to thank my other committee members, Prof. Deepak Ganesan, Prof. James Kurose, and Prof. Michael Zink, who have constantly given me valuable inputs on various research projects, that led to many high quality publications.

I would like to thank my colleagues Jeremy Gummeson, Pranshu Sharma, Sean Barker, and Dilip Krishnappa with whom I have worked on several projects which form part of this dissertation. I am also grateful to the other members of the LASS group, particularly Upendra Sharma and Rahul Singh, with whom I have spent countless hours discussing a range of topics from research to politics. My time in Amherst was made even more enjoyable by the incredible group of friends I developed over the years. Specifically, I am grateful to Ameet Shetty, Anindya Misra, JP Mahalik, Supratim Mukherjee, and Vimal Mathew, for always being ready to help whenever I needed any.

My sincere thanks to Tyler Trafford for his assistance in managing the machines that were central to my doctoral work. Many thanks to the CSCF staff who helped me overcome several technical problems that I faced while developing research prototypes for my thesis. I am also indebted to Leeanne Leclerc for keeping my paperwork organized and on track, and to Karren Sacco for handling all administrative work; they have taken care of many non-technical problems I had.

This journey would not have been started without the support and guidance of my undergraduate advisors – Prof. Arobinda Gupta, Prof. Rajarshi Roy, and Prof. Shamik Sural – and my mentor D. D. Ganguly. They have always believed in me and inspired me to dream beyond my limitations. I am also fortunate to have worked with an awesome group of people before starting my PhD career, particularly my supervisor Rohit Shankar and colleagues Bharat Varma, Prasanth Radhakrishnan, and Rajesh Dharmalingam, whose friendships I will always cherish.

Finally, I am obliged to my parents, my brothers Santosh Sharma and Mukesh Sharma, my sisters-in-law Nitu Sharma and Nupur Sinha, and my nephew Nitin Sharma, who have influenced me the most and were always there to support and encourage me. They have been my source of strength. I dedicate this thesis to them.
ABSTRACT

DESIGNING DISTRIBUTED SYSTEMS FOR INTERMITTENT POWER

MAY 2013

NAVIN KUMAR SHARMA

B.Tech., INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR INDIA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Prashant Shenoy

The increasing demand for computing infrastructure, such as data centers and storage systems, has increased their energy footprint. As a result of this growth, computing infrastructure today contribute 2-3% of the global carbon emissions. Furthermore, the energy-related costs have now become a significant fraction of the total cost of ownership (TCO) of a modern computing infrastructure. Hence, to reduce the financial and environmental impact of growing energy demands the design of eco-friendly green infrastructure has become an important societal need. This thesis focuses on designing distributed systems, primarily data centers and storage systems, to run on renewable energy sources such as solar and wind.

As renewables are intermittent in nature, accurate predictions of future energy is important for a distributed system to balance workload demand and energy supply, and optimize its performance amid significant and frequent changes in both demand and supply. To accurately predict energy harvesting in all weather conditions, I develop two prediction models that leverage weather forecasts to predict solar and wind energy harvesting. The first prediction model is an empirical model that uses sky cover forecast and wind speed forecast to predict solar energy and wind energy, respectively, in the future. The second prediction model is a machine learning based model that uses statistical power of machine learning techniques to give better predictions of solar energy harvesting than the empirical model.

To regulate the energy footprint of a server I propose a new energy abstraction, called Blink, that applies duty cycle to the server to cap power consumption to supply. I also pro-
pose several blinking policies to coordinate blinking across servers to regulate cluster-wide power consumption with changes in the available power. Further, I show that a real-world application can be redesigned, with modest complexity, to perform well on intermittent power.

To extend the applicability of blinking beyond an in-memory cache server I use the blinking abstraction to design two different distributed systems – (a) Distributed File System, and (b) Multimedia Cache – for intermittent power. I propose several design techniques, including a staggered blinking policy and power-balanced data layout, to optimize the performance of these systems under intermittent power scenarios. Additionally, I experiment with three unmodified real-world applications – (a) Memcache, (b) MapReduce, and (c) Search Engine – to test the practicality of our blink-aware file system. Our results show that real-world applications can perform reasonably well for real workloads in spite of significant and frequent variations in power supply. Finally, I use a real WiMAX testbed to demonstrate that our blink-aware multimedia cache can significantly save bandwidth usage of cell towers while providing good performance under intermittent power constraints.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGMENTS</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xii</td>
</tr>
</tbody>
</table>

## CHAPTER

1. **INTRODUCTION**                                      | 1   |
   1.1 Background and Motivation                         | 1   |
   1.2 Thesis Contributions                              | 3   |
       1.2.1 Energy Harvesting Prediction                  | 4   |
       1.2.2 Energy Footprint Regulation                   | 5   |
       1.2.3 Performance Optimization                     | 6   |
   1.3 Thesis Outline                                    | 6   |

2. **BACKGROUND AND RELATED WORK**                      | 8   |
   2.1 Energy Harvesting Prediction                      | 8   |
   2.2 Power Management in Data Centers                  | 8   |
   2.3 Duty Cycling in Low-Power Devices                 | 10  |
   2.4 Distributed File Systems                          | 10  |
   2.5 Multimedia Caches                                 | 11  |

3. **WEATHER FORECASTS BASED ENERGY HARVESTING**        | 12  |
   3.1 Background and Motivation                         | 12  |
   3.2 The Case for Using Forecasts                      | 14  |
   3.3 Forecast → Energy Model                           | 17  |
       3.3.1 Sky Condition → Solar Power Model            | 18  |
       3.3.1.1 Computing Solar Power From Solar Radiation | 18  |
       3.3.1.2 Computing the Maximum Possible Solar Power | 19  |
       3.3.1.3 Solar Model                                 | 20  |
3.3.2 Wind Speed \rightarrow Wind Power Model \rightarrow 20
3.3.3 Compensating for Forecast Errors \rightarrow 21
3.4 Comparison with PPF Variants \rightarrow 21
3.5 Related Work \rightarrow 24
3.6 Conclusion \rightarrow 24

4. MACHINE LEARNING MODEL FOR SOLAR ENERGY HARVESTING
PREDICTION \rightarrow 25
4.1 Background and Motivation \rightarrow 25
4.2 Data Analysis \rightarrow 26
4.3 Prediction Models \rightarrow 29
4.3.1 Linear Least Squares Regression \rightarrow 30
4.3.2 Support Vector Machines \rightarrow 31
4.3.3 Eliminating Redundant Information \rightarrow 33
4.3.4 Comparing with Existing Models \rightarrow 33
4.4 Conclusion \rightarrow 34

5. MANAGING SERVER CLUSTERS ON INTERMITTENT POWER \rightarrow 35
5.1 Background and Motivation \rightarrow 35
5.1.1 Example: BlinkCache \rightarrow 36
5.1.2 Contributions \rightarrow 37
5.2 Blink: Rationale and Overview \rightarrow 38
5.3 Blink Prototype \rightarrow 41
5.3.1 Blink Hardware Platform \rightarrow 42
5.3.1.1 Energy Sources \rightarrow 42
5.3.1.2 Low-power Server Cluster \rightarrow 43
5.3.2 Blink Software Architecture \rightarrow 44
5.4 Blinking Memcached \rightarrow 45
5.4.1 Memcached Overview \rightarrow 45
5.4.2 Access Patterns and Performance Metrics \rightarrow 46
5.4.3 BlinkCache Design Alternatives \rightarrow 47
5.4.3.1 Activation Policy \rightarrow 48
5.4.3.2 Synchronous Policy \rightarrow 50
5.4.3.3 Load-Proportional Policy \rightarrow 51
5.4.4 Summary \rightarrow 52
5.5 Implementation and Evaluation \rightarrow 52
5.5.1 Benchmarks \rightarrow 53
5.5.1.1 Activation Blinking and Thrashing \rightarrow 55
5.5.1.2 Synchronous Blinking and Fairness \rightarrow 56
5.5.1.3 Balancing Performance and Fairness \rightarrow 57
5.5.2 Case Study: Tag Clouds in GlassFish \rightarrow 59
5.6 Related Work \rightarrow 60
5.7 Conclusion \rightarrow 61
6. DISTRIBUTED FILE SYSTEM FOR INTERMITTENT POWER .......... 63
   6.1 Background and Motivation ........................................ 63
   6.2 DFSs and Intermittent Power ....................................... 66
      6.2.1 Energy-Proportional DFSs ..................................... 67
      6.2.2 Migration-based Approach ..................................... 68
      6.2.3 Equal-Work Approach .......................................... 69
   6.3 Applying Blinking to DFSs ......................................... 70
   6.4 BlinkFS Design ....................................................... 70
      6.4.1 Reading and Writing Files ..................................... 72
      6.4.2 Reducing the Latency Penalty .................................. 74
   6.5 Implementation ....................................................... 76
   6.6 Evaluation ............................................................ 79
      6.6.1 Benchmarks ....................................................... 79
      6.6.2 Case Studies ................................................... 81
   6.7 Conclusion ........................................................... 84

7. MULTIMEDIA CACHE FOR INTERMITTENT POWER ........................ 85
   7.1 Background and Motivation ......................................... 85
   7.2 Cache and Intermittent Power ....................................... 87
   7.3 GreenCache Feasibility: Trace Analysis ............................ 90
   7.4 GreenCache Design .................................................. 92
      7.4.1 Minimizing Bandwidth Cost .................................... 93
      7.4.2 Reducing Buffering Time ....................................... 94
         7.4.2.1 Staggered Load-Proportional Blinking .................. 94
         7.4.2.2 Prefetching Recommended Videos .......................... 95
   7.5 GreenCache Implementation ......................................... 96
   7.6 Experimental Evaluation ............................................ 98
      7.6.1 Benchmarks ..................................................... 98
      7.6.2 Staggered load-proportional blinking ......................... 100
      7.6.3 Case study ..................................................... 102
   7.7 Related Work ......................................................... 102
   7.8 Conclusion ........................................................... 103

8. SUMMARY AND FUTURE WORK ........................................ 104
   8.1 Thesis Summary ...................................................... 104
   8.2 Future Work .......................................................... 105

BIBLIOGRAPHY .............................................................. 107
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Challenges with existing green computing techniques for data centers.</td>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
<td>Values for a, b, and c in our quadratic solar power model, which is a</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>function of the time of day for each month of the year.</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Correlation matrix showing correlation between different forecast</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>parameters.</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Latencies for several desktop and laptop models to perform a complete S3</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>cycle (suspend and resume). Data from both [35] and our own measurements of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Blink’s OLPC-X0.</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Blink APIs for setting per-node blinking schedules.</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Blink’s measurement APIs that applications use to inform their blinking</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>decisions.</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Summary of the best policy for a given performance metric and workload</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>combination.</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>POSIX-compliant API for BlinkFS.</td>
<td>77</td>
</tr>
<tr>
<td>6.2</td>
<td>Standard deviation and 90th percentile latency.</td>
<td>80</td>
</tr>
<tr>
<td>7.1</td>
<td>Standard deviation, 90th percentile, and average buffering time.</td>
<td>98</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>System overview</td>
</tr>
<tr>
<td>3.1</td>
<td>Power generated during a 12 day period in October, 2009 from our solar panel (a) and wind turbine (b).</td>
</tr>
<tr>
<td>3.2</td>
<td>The error in sky condition (a) and wind speed (b) when using the past to predict the future for different time intervals in 2008 at 1 hour and 5 minute granularities, respectively, for Amherst, Massachusetts.</td>
</tr>
<tr>
<td>3.3</td>
<td>RMSE between the observed sky condition and wind speed and those predicted by NWS forecasts from 3 hours to 72 hours in the future.</td>
</tr>
<tr>
<td>3.4</td>
<td>Relationship between the solar radiation our weather station observes and the power generated by our solar panel.</td>
</tr>
<tr>
<td>3.5</td>
<td>Profile for solar power harvested on clear and sunny days in January, May, and September, and the quadratic functions $f(x)$, $g(x)$, and $h(x)$ we fit to each profile, respectively.</td>
</tr>
<tr>
<td>3.6</td>
<td>Power output from our wind turbine and the power output predicted by our wind power model over the first 3 weeks of October. The graph shows the rated power curves from the wind turbines manual for steady and turbulent wind, as well as our fitted curve.</td>
</tr>
<tr>
<td>3.7</td>
<td>Power output from our solar panel and the power output predicted by different prediction models over the first 3 weeks of October, 2009.</td>
</tr>
<tr>
<td>3.8</td>
<td>Power output from our wind turbine and the power output predicted by different prediction models over the first 3 weeks of October.</td>
</tr>
<tr>
<td>4.1</td>
<td>Solar intensity shows seasonal variation with days of a year, although daily weather conditions also have a significant impact.</td>
</tr>
<tr>
<td>4.2</td>
<td>Solar intensity and wind speed show little correlation (a). Solar intensity shows some correlation with temperature at high temperatures (b) and with dew point at high dew points (c).</td>
</tr>
</tbody>
</table>
4.3 Solar intensity generally decreases with increasing values of sky cover (a), relative humidity (b), and precipitation potential (c). ................................. 28

4.4 Relative humidity (a) and precipitation % (b) positively correlate with sky cover. Relative humidity also increases with increasing precipitation % (c). ......................................................... 29

4.5 Observed and predicted solar intensity using linear least squares regression for September and October 2010. ................................. 31

4.6 Observed and predicted solar intensity, using SVM regression with an RBF kernel, for the months of September and October 2010. .............. 32

4.7 Observed and predicted solar intensity, using three different prediction techniques — (a) SVM-RBF kernel with 4 dimensions, (b) cloudy computing model using sky condition forecast, (c) past predicts future prediction model — for the months of September and October 2010. ................................................................. 32

5.1 The popularity of web data often exhibits a long heavy tail of equally unpopular objects. This graph ranks the popularity of Facebook group pages by their number of fans. ......................................................... 37

5.2 Hardware architecture of the Blink prototype. ................................. 41

5.3 Empirically-measured battery capacity as a function of voltage for our deep-cycle battery. We consider the battery empty below 12V, since using it beyond this level will reduce its lifetime. ................................. 43

5.4 The popularity rank, by number of fans, for all 20 million public group pages on Facebook follows a Zipf-like distribution with $\alpha = 0.6$. ................................. 46

5.5 To explicitly control the mapping of keys to servers, we interpose always-active request proxies between memcached clients and servers. The proxies are able to monitor per-key hit rates and migrate similarly popular objects to the same nodes. ................................. 49

5.6 Graphical depiction of a static/dynamic activation blinking policy (a), an activation blinking policy with key migration (b), and a synchronous blinking policy (c). ......................................................... 51
5.7 The near 2 second latency to transition into and out of S3 in our prototype discourages blinking intervals shorter than roughly 40 seconds. With a 50% duty cycle we expect to operate at 50% full power, but with a blink interval of less than 10 seconds we operate near 100% full power.

5.8 Maximum throughput (a) and latency (b) for a dedicated memcached server, our memcached proxy, and a MySQL server. Our proxy imposes only a modest overhead compared with a dedicated memcached server.

5.9 Under constant power for a Zipf popularity distribution, the dynamic variant of the activation policy performs better than the static variant as power decreases. However, the activation policy with key migration outperforms the other variants.

5.10 Under oscillating power for a Zipf popularity distribution, the static variant of the activation policy performs better than the dynamic variant as the oscillation increases. Again, the activation policy with key migration outperforms the other variants.

5.11 For a uniform popularity distribution, both the synchronous policy and the dynamic variant of the activation policy are significantly more fair, i.e., lower standard deviation of average per-object latency, than the activation policy with key migration.

5.12 For a uniform popularity distribution, the synchronous policy and the activation policy with key migration achieve a similar hit rate under different power levels. Both policies achieve a better hit rate than the dynamic variant of the activation policy.

5.13 The load-proportional policy is more fair to the unpopular objects, i.e. bottom 80% in popularity, than the activation policy with key migration for Zip popularity distributions, especially in low-power scenarios.

5.14 The load-proportional policy has a slightly lower hit rate than the activation policy with key migration.

5.15 As S3 transition overhead increases, the hit rate from the load-proportional policy decreases relative to the activation policy with key migration for a Zipf distribution at a moderate power level.

5.16 Power signal from a combined wind/solar deployment (a) and average page load latency for that power signal (b).
6.1 Electricity prices vary every five minutes to an hour in wholesale markets, resulting in the power available for a fixed monetary budget varying considerably over time. ................................. 64

6.2 Inaccessible data rises with the fraction of inactive nodes using a random replica placement policy. ................................. 68

6.3 Simple example using a migration-based approach (a) and blinking (b) to deal with power variations. ................................. 69

6.4 BlinkFS Architecture ................................. 71

6.5 Combining staggered blinking (a) with a power-balanced data layout (b) maximizes block availability. ................................. 72

6.6 BlinkFS hardware prototype. ................................. 78

6.7 Maximum sequential read/write throughput for different block sizes with and without the proxy. ................................. 78

6.8 Read and write latency in our BlinkFS prototype at different power levels and block replication factors. ................................. 79

6.9 Maximum possible throughput in our BlinkFS prototype for different number of block servers. ................................. 81

6.10 BlinkFS performs well as power oscillation increases. ................................. 82

6.11 MapReduce completion time at steady power levels and using our combined wind/solar power trace. ................................. 82

6.12 MemcacheDB average latency at steady power levels and using our combined wind/solar power signal. ................................. 83

6.13 Search engine query rate with price signal from 5-minute spot price in New England market. ................................. 84

7.1 Solar and wind energy harvesting from our solar panel and wind turbine deployment on three consecutive days in Sep 2009. ................................. 88

7.2 The top part of the figure shows a potential streaming schedule for a blinking node while the bottom half shows the smooth play out with is achieved with the aid of a client-side buffer. ................................. 90

7.3 Video Popularity (100 out of 105339) ................................. 91
7.4 Related Video Position Analysis .................................................. 92
7.5 Video Switching Time Analysis ................................................. 92
7.6 GreenCache Architecture ......................................................... 93
7.7 Staggered load-proportional blinking ........................................ 93
7.8 Hardware Prototype ............................................................... 96
7.9 Both bandwidth usage and buffering time reduce with increasing cache size ................................................................. 99
7.10 Video chunking reduces both bandwidth usage and buffering time. .... 99
7.11 Buffering time at various steady and oscillating power levels .......... 100
7.12 Buffering time decreases as the number of prefetched videos (first chunk only) from related lists increases. .............................. 100
7.13 Buffering time at various power levels for our combined solar/wind power trace ................................................................. 102
CHAPTER 1
INTRODUCTION

As distributed systems, such as data centers and storage systems, are growing in numbers and sizes the energy demands of these systems are also growing. The growing energy demands of distributed systems lead to burning more fossil fuels, which has long-term financial and environmental impact on our society. Reducing dependency on dirty fossil fuels by leveraging renewables to power distributed systems has become an important societal need. This thesis addresses challenges associated with running distributed systems on renewables, or intermittent power in general, and presents how distributed systems can be designed to operate completely off renewables while performing well at all power levels.

1.1 Background and Motivation

As computing infrastructure today contribute a significant fraction of the global carbon emissions, both industry and the research community are giving serious consideration on reducing the energy footprint of distributed systems. One obvious way to reduce the energy footprint of distributed systems is to make them energy-proportional and more energy-efficient. In addition, one could also use a combination of renewables, such as solar and wind, and fossil fuels to power distributed systems, and focus on designing techniques to completely run the distributed systems off the renewables. The benefits of using renewables are twofold. First, renewables are eco-friendly and don’t have any adverse environmental impact. Second, with increasing research attention on designing more sophisticated and energy-efficient conversion technologies to generate electricity from renewables, in the future, the cost of renewable energy is likely to become cheaper than that of grid energy.

Prior research assumes that data centers have an unlimited supply of power, and focuses largely on optimizing applications to use less energy without impacting performance. One such approach is to consolidate loads on a small number of servers, e.g., using request redirection or VM migration, and deactivate the remaining servers during off-peak hours. Since many applications, such as MapReduce, maintain distributed states, this approach necessitates the migration of states before deactivating the servers, which might become prohibitively expensive in a typical data center.

Another approach for reducing energy consumption in data centers is to use a heterogeneous set of nodes – low-power nodes along with high-power nodes [35, 36, 49]. These techniques maintain a set of always-on low-power nodes to coexist with a set of high-power nodes, which are sent to a low-power state during idle periods or off-peak hours. The low-power nodes maintain active connections and provide basic services, such as responding to ping requests, ICMP and ARP packets, etc., and wake up the high-power nodes on demand.
Though these approaches reduce the idle power waste, a significant energy saving might be hard to achieve if idle periods are not long enough to compensate for state transition overheads of the high-power nodes. Unlike mobile devices, high-power servers take several seconds to sleep, and thus cannot sleep during short idle periods, which are frequent for many applications [127].

In recent years, the research community has started looking into various ways of integrating renewables to partially power data centers, while still relying on grids to provide the backup power [48, 50, 53, 70, 111, 110, 109]. Increasing renewable penetration demands adaptation to a variable power supply. While these techniques try to maximize the renewable penetration without degrading application performance, they are not designed to handle intermittent power constraints, such as significant and frequent variations in power supply or a sustained low power scenario. In contrast, there has been little research on running data centers solely on renewables, and optimizing application performance for intermittent power that fluctuates over time. The primary challenge with using renewables is that they are intermittent in nature. Consequently, a system should be designed to handle significant and frequent changes in the available power. Moreover, simply designing a system to operate on intermittent power would not suffice because the performance might be intolerable in many cases. For example, a storage system could spend hours migrating data to prevent data loss when the available power drops down significantly (∼90%), and thus becoming inaccessible for a long duration. So, running distributed systems solely off renewables would be practical only when they adapt to intermittency in the power supply while performing well in spite of significant and frequent variations in the available power.

Aforementioned techniques could also use a large energy storage, e.g., a large number of battery arrays, to handle intermittent power constraints. But, accurate predictions of workload demand and energy supply are required to smooth out the power supply. If predictions are inaccurate or there is a mismatch between supply and demand predictions, even a large energy storage would not prevent intermittency in the power supply. Further, maintaining a large energy storage is unsustainable and prohibitively expensive. Designing systems for renewables, or intermittent power in general, requires a dramatic departure from reducing energy consumption in distributed systems running on uninterrupted power supply. The former demands optimizing performance for given power supply, while the latter focuses on minimizing energy consumption for given workload demands or performance SLAs.

The central theme of this thesis is to explore how distributed systems can handle extreme power constraints, ranging from very high fluctuations to very low power scenarios, while performing well at all power levels without any knowledge of future supply and demand. Since a large energy storage is unsustainable and prohibitively expensive to manage, this thesis focuses on designing systems that adapt to the power supply, i.e., instantly regulate energy footprint to match energy demand with supply. As renewables are intermittent and workload demands vary independent of harvested energy, accurate energy predictions can significantly improve the performance without requiring a large energy storage. For example, a storage system could perform costly data migration in advance, when the power is plenty, to keep data available when the power becomes scarce. Consequently, designing distributed systems to operate off renewables, or intermittent power in general, consists of three major parts.
### Table 1.1. Challenges with existing green computing techniques for data centers.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activation+Migration</td>
<td>high migration overhead</td>
</tr>
<tr>
<td></td>
<td>inaccessible data at low power</td>
</tr>
<tr>
<td>Heterogenous nodes</td>
<td>no energy savings during short idle periods</td>
</tr>
<tr>
<td></td>
<td>cannot handle intermittent power constraints</td>
</tr>
<tr>
<td>Renewables+Grid</td>
<td>difficult to handle grid interruptions</td>
</tr>
<tr>
<td></td>
<td>cannot leverage variable electricity pricing</td>
</tr>
</tbody>
</table>

1. **Energy Harvesting Prediction.** Designing techniques to accurately predict future energy harvesting at short-to-medium time scales.

2. **Energy Footprint Regulation.** Designing hardware and software interface to regulate energy footprint of distributed systems to cap power demand to power supply.

3. **Performance Optimization.** Designing techniques to maximize application performance at all power levels, and using future energy predictions for scheduling operations to optimize overall performance of the system.

This thesis focuses primarily on data centers and storage systems as they are particularly well-positioned to benefit from renewables, since unlike household and industrial loads, delay-tolerant batch workloads may permit performance degradation due to varying power.

### 1.2 Thesis Contributions

*The fundamental thesis of this dissertation is to demonstrate that distributed systems can be redesigned to run solely on renewables, or intermittent power in general, while performing well in spite of significant and frequent variations in the available power.* To this end I have designed five key systems (Figure 1.1) which provide new algorithms and mechanisms to run distributed systems on intermittent power while maximizing application performance at all power levels.

- **Cloudy Computing.** A novel technique to predict future solar and wind energy harvesting from weather forecasts [94].

- **ML Forecast.** A powerful statistical model to predict solar energy harvesting using machine learning techniques [95].

- **Blink.** A new energy abstraction for regulating energy footprint of server clusters in order to cap power consumption to the available power [93].

- **BlinkFS.** A distributed file system for intermittent power that avoids costly data migration and maximizes I/O throughput and latency at all power levels [97].
Figure 1.1. The systems presented in this thesis to design distributed systems for intermittent power.

- **GreenCache.** A multimedia cache for intermittent power that minimizes bandwidth cost and access latency at all power levels [96].

### 1.2.1 Energy Harvesting Prediction

As discussed above, having knowledge of future energy harvesting is important for many distributed systems, running off renewables, to optimize application performance. Although many techniques exist to forecast future energy harvesting, they fail to predict drastic changes in weather condition because their predictions are based on observations in the past. While the past is accurate for both sufficiently short, i.e., seconds to minutes, and sufficiently long, i.e., months to years, time-scales, I show in Chapter 3 that predictions based on weather forecasts are more accurate at the medium time-scales, i.e., hours to days, relevant to a large class of distributed systems. Further, since forecasts from the National Weather Service (NWS) are based on complex mathematical models and observations from satellites and high-end radars, they tend to be reasonably accurate at the medium time-scales. Moreover, NWS forecasts can also predict drastic changes in weather condition, e.g., an incoming thunderstorm, which is not possible from predictions based on the past.

To improve the accuracy of energy harvesting prediction amid significant weather changes, I derive empirical models in Chapter 3 that leverage sky cover and wind speed forecast from NWS to predict solar and wind energy harvesting, respectively, at short to medium-length intervals, i.e., hours to days. I also compare the efficiency of our prediction models with state-of-the-art prediction techniques, and show that forecast-based models outperform the existing techniques at short to medium time intervals.

As we expect, different weather parameters do not vary independently; rather they are correlated with each other. For example, if the temperature or dew point is high, then the solar intensity is likely to be high. But, understanding relationships between different weather parameters and how they affect each other is not possible with simple empirical models, as described above. In Chapter 4 I explore machine learning techniques to understand relationships between different weather parameters and how they affect solar intensity. I find that sky cover is not the only parameter which affects solar intensity; in fact, many other parameters, such as humidity and precipitation, have strong correlation with solar intensity. To leverage all forecast parameters in solar energy prediction, I design machine learning
models that predict solar energy harvesting based on time and forecast of all weather parameters. Our results show that SVM-based prediction models built using seven distinct weather forecast metrics outperform aforementioned forecast-based models by more than 20% for our site.

1.2.2 Energy Footprint Regulation

Unlike grid energy, renewables are intermittent and can vary significantly and frequently without any warning. To operate completely off renewables distributed systems must be able to regulate their power consumption to match the available power. Designing energy-proportional systems is challenging since many server components, including the CPU, memory, disk, motherboard, and power supply, now consume significant amounts of power. Thus, any power optimization that targets only a single component is not sufficient, since it reduces only a fraction of the total power consumption. For example, a modern server that uses dynamic voltage and frequency scaling in the CPU at low utilization may still operate at over 50% of its peak power. Thus, deactivating entire servers, including most of their components, remains the most effective technique for controlling energy consumption in server farms, especially at low power levels that necessitate operating servers well below 50% peak power on average.

Simply varying the number of active nodes in response to changes in power has several drawbacks. First, application states might get lost if they are not transferred to active nodes before deactivating a set of nodes. Since power variations may be significant, it might not be possible to transfer all states before deactivating the nodes. Second, if the available power is low (~20% of peak power) consolidating all states on a small set of active nodes might not be possible, since there might not be enough space to store the states. Third, state transfer might be prohibitively expensive in a typical data center for even small variations in power supply. For example, in a typical data center of 1000 nodes, where each node stores 500 GB of data, deactivating 20% of the nodes, for 20% drop in the power, would lead to a 100 terabyte migration in the worst case. Migrating this data over a ten gigabit link would take more than 10 hours, and prevent the data center from performing useful work.

Since long-term battery-based storage is prohibitively expensive, increasing renewable integration requires closely matching power consumption to supply. For instant regulation of the energy footprint of a server cluster, in response to changes in the power supply, in Chapter 5 I propose a new energy abstraction, called Blink, for gracefully handling intermittent power constraints. Blinking applies a duty cycle to servers that controls the fraction of time they are in the active state, e.g., by activating and deactivating them in succession, to gracefully vary their energy footprint. Blinking generalizes the extremes of either keeping a server active (a 100% duty cycle) or inactive (a 0% duty cycle) by providing a spectrum of intermediate possibilities. I also propose a number of blinking policies to coordinate blinking of servers in a server cluster. I then design an application-independent platform for developing and evaluating blinking applications, and use it to perform an in-depth study of the effects of blinking on one particular application and power source: Memcached operating off renewable energy. Our results show that a real-world distributed application can
be redesigned, with modest overhead, to perform well on a server cluster operating solely off renewables.

1.2.3 Performance Optimization

As discussed above, Blink provides an application-independent abstraction for regulating energy footprint of a server. Further, blinking policies provide different ways of coordinating blinking of servers in a server cluster to regulate the energy footprint of the cluster. Blinking is a general abstraction for leveraging intermittent power in a wide range of applications. In Chapter 5 I discuss the applicability of the blinking abstraction for an in-memory distributed cache server. Another important application that can leverage the blinking abstraction is a DFS (distributed file system). Since DFSs now serve as the foundation for a wide range of data-intensive applications run in today’s data centers, designing a DFS optimized for intermittent power opens avenues for a number of applications to leverage the blinking abstraction without any modification. But designing such a DFS poses a significant research challenge, since periods of scarce power may render data inaccessible, while periods of plentiful power may require costly data layout adjustments to scale up I/O throughput.

Chapter 6 presents the design of BlinkFS, a distributed file system for intermittent power, and shows how traditional file system operations can be redesigned to handle intermittency in the power supply. Further, I propose several techniques, including a staggered blinking policy and power-balanced data layout, to optimize I/O throughput and latency as power varies. To see the feasibility of BlinkFS for real-world applications I experiment with three unmodified real-world applications – Memcache, MapReduce, Search Engine – on a small cluster of 10 Mac minis. Our results show that, with proposed optimizations, these real-world applications perform reasonably well in spite of significant and frequent variations in power supply.

Unlike an in-memory key-value store, multimedia caches store data on persistent storage which become inaccessible at low power. Traditional multimedia caches assume unlimited supply of power and focus primarily on reducing backhaul bandwidth usage. But running a multimedia cache on intermittent power warrants new approaches as a period of low power makes cached contents unavailable, in which case reducing the bandwidth cost increases buffering time for end users. Thus, designing multimedia caches for intermittent power poses several new research challenges. In Chapter 7 I present the design of a distributed multimedia cache for intermittent power, called GreenCache, as well as several optimization techniques to provide multimedia caching while minimizing bandwidth cost and latency at all power levels.

1.3 Thesis Outline

This thesis is structured as follows. Chapter 2 provides background on green computing techniques in data centers and storage systems. Chapter 3 describes energy harvesting prediction models that leverage weather forecasts to predict solar and wind energy for any forecast interval. This is followed in Chapter 4 with use of machine learning techniques to
design better prediction models for solar energy harvesting prediction. Chapter 5 presents a hardware/software design to run server clusters on renewables, or intermittent power in general. Chapter 6 presents the design of a blink-aware distributed file system to manage stateful applications running concurrently on a data center running off renewables. Chapter 7 leverages the blinking abstraction from Chapter 5 to design a multimedia cache for intermittent power. Finally, Chapter 8 concludes the thesis and provides a brief overview of future work.
CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter provides a brief overview of energy harvesting prediction techniques and green computing techniques to reduce energy footprint of data centers and storage systems. Subsequent chapters also provide more detailed related work sections.

2.1 Energy Harvesting Prediction

Much of the prior work on energy harvesting assumes that past observed data provides a good indication of future energy harvesting. The simplest of these techniques is the past-predicts-future (PPF) model that predicts future energy harvesting as energy harvested in the immediate past [9, 13]. Many variants of the PPF model have been proposed in recent years. Kansal et al. [9] maintain an exponentially weighted moving average (EWMA) for solar power to achieve energy-neutral operation in a system with elastic workload demands. The EWMA approach is a variant of PPF that adapts to seasonal variations in solar radiation. However, EWMA does not account for drastic changes in weather conditions, which are frequent in many areas. Noh et al. [15] use a historical model for solar radiation that maintains an expectation for each time slot in a day based on the previous day’s solar radiation reading, but down-scales all future time-slots in a day by $N\%$ if it records a solar radiation reading $N\%$ less than expected.

The primary drawback of PPF techniques is that they fail to predict solar energy when weather changes unpredictably and inconsistently. Furthermore, PPF techniques do not apply to wind speed or wind power predictions since wind is more intermittent than solar and not diurnal in nature. We know of no work that discusses prediction strategies for wind speed. The recent commoditization and emergence of micro-wind turbines, such as the 400 watt Air-X we use in our deployment, motivates further study of harnessing wind power in sensor systems deployed at locations with ample wind but little sunlight, i.e., during the winter in the extreme north or south. As I discuss in Chapter 3, weather forecasts are a better predictor of future energy harvesting than the immediate past at medium time-scales ranging from 3 hours to 3 days for both solar and wind.

2.2 Power Management in Data Centers

Prior work on reducing energy footprint of data centers has largely been focused on minimizing energy consumption for given workload demand or SLAs. Techniques include consolidating load onto a small number of servers, e.g., using request redirection [49, 41]
or VM migration, and powering down the remaining servers during off-peak hours, or balancing load to mitigate thermal hotspots and reduce cooling costs [37, 60, 61]. Another class of techniques, including SleepServer [36], uses a heterogeneous set of nodes (low-power nodes in coexistence with high-power nodes) to reduce idle power waste without degrading application performance. These techniques transition the high-power nodes to a low-power state, such as ACPI S3 state, to conserve power during idle periods. Since, unlike mobile devices, high-power servers take several seconds to sleep, transitioning them to the sleep state might not be possible during short idle periods.

Power capping has also been studied in data centers to ensure collections of nodes do not exceed a worst-case power budget [67, 45]. However, the work assumes exceeding the power budget is a rare transient event that does not warrant application-specific modifications, and that traditional power management techniques, e.g., DVFS, are capable of enforcing the budget. These assumptions may not hold in many scenarios with intermittent power constraints, as with renewable energy power source. Intermittent power constraints are also common in developing regions that experience “brownouts” where the electric grid temporarily reduces its supply under high load. BrownMap [75] proposes a technique that uses automatic VM resizing and live migration to maximize the performance while dealing with power outages in a shared data center. Price-driven optimizations, due to either demand-response incentives or market-based pricing, introduce intermittent constraints as well, e.g., if multiple data centers coordinate to reduce power at locations with high spot prices and increase power at locations with low spot prices [66].

The increasing energy consumption of data centers [34] has led companies to invest heavily in renewable energy sources [59, 72]. Both Microsoft (at the recent Rio+20 summit) [105] and HP [109] have announced bold initiatives to design net-zero data centers that consume no net energy from the electric grid and include substantial use of on-site renewable energy sources. Additionally, startups, such as AISO.net [104], have formed around the idea of green hosting using only renewables. Researchers have also started to design green data centers partially or completely powered by renewables. Goiri et al. [53] have designed Parasol, a solar-powered micro-datacenter. They intelligently schedule workloads and energy sources to significantly reduce the energy cost. Liu et al. [100] have also proposed a novel approach to integrate energy supply and cooling supply with IT workload planning to improve the performance of data center operations while minimizing the energy cost.

Since long-term battery-based storage is prohibitively expensive, increasing renewable penetration requires closely matching power consumption to generation. Past work on green data centers often assumes that grid energy is always available [53, 110, 111], and focuses largely on optimizing applications to use less energy without impacting performance. In contrast, running data centers solely on renewables warrants optimizing performance for intermittent power that fluctuates over time. The key challenge in designing data centers for renewables is optimizing application performance in the presence of power constraints that may vary significantly and frequently over time. Importantly, these power and resource consumption constraints are independent of workload demands. In Chapter 5, I discuss a new energy abstraction to regulate energy footprint of servers to match available power, and design several techniques to maximize applications’ performance at all power levels.
2.3 Duty Cycling in Low-Power Devices

The sensor network community has extensively studied strategies for dealing with variable sources of renewable power, since these systems often do not have access to the power grid. They often use duty cycling nodes to reduce energy consumption and prolong battery life of low-power sensor nodes [79, 80, 81]. Since these nodes rely on environmental energy to support perpetual operations, they use intelligent duty-cycle mechanisms to cap the energy consumption to the harvested energy. Similarly, mobile computing generally focuses on extending battery life by regulating power consumption [77], rather than modulating performance to match energy production.

2.4 Distributed File Systems

Distributed file systems (DFSs), such as the Google File System (GFS) [122] or the Hadoop Distributed File System (HDFS) [68], distribute file system data across multiple nodes. Designing energy-proportional DFSs is challenging, in part, since naively deactivating nodes to reduce energy usage has the potential to render data inaccessible [126]. One simple way to prevent data on inactive nodes from becoming inaccessible is by storing replicas on active nodes. Replication is already used to increase read throughput and reliability in DFSs, and is effective if the fraction of inactive nodes is small. However, the fraction of inaccessible data rises dramatically when the power drops down below 50%, even for aggressive replication factors, such as 7.

One popular approach to deal with power variations is migration based approach that varies power consumption by migrating data to concentrate it on a set of active nodes, and then deactivating the remaining nodes [48, 65, 78]. With this approach, the migration overhead becomes prohibitively expensive when power varies significantly. For example, in a data center of thousand nodes even a 2% reduction in the available power would necessitate deactivation of twenty nodes. Assuming each node stores 500GB of data, 10TB of data needs to be migrated which would take more than an hour even on a ten gigabit link.

Another approach for designing energy-efficient storage systems is to use concentrated data layouts, which deactivate nodes without causing inaccessible data. The layouts often store primary replicas on one subset of nodes, secondary replicas on another mutually-exclusive subset, tertiary replicas on another subset, etc., to safely deactivate non-primary nodes [126]. Amur et al. [38] propose an energy-proportional DFS, called Rabbit, that eliminates migration-related thrashing using an equal-work data layout. The layout uses progressively larger replica sets, e.g., more nodes store \((n + 1)\)-ary replicas than \(n\)-ary replicas. Specifically, the layout orders nodes 1...\(i\) and stores \(b_i = \frac{B}{i}\) blocks on the \(i\)th node, where \(i > p\) and \(p\) nodes store primary replicas (assuming a data size of \(B\)). Other systems concentrate data to optimize for skewed access patterns, by storing only popular data on a small subset of active nodes [116, 121, 48, 65, 78]. Unfortunately, concentrated layouts cause problems if available power varies independently of workload demands. Further, even the primary replicas become inaccessible in sustained low-power scenarios.

As I discuss in Chapter 6, BlinkFS avoids costly data migration even if the available power changes significantly and frequently. Additionally, the blinking approach is ben-
eficial at low power levels if not enough nodes are active to store all data, since data is accessible for some period each blink interval. BlinkFS employs several techniques to reduce I/O latency at low power with modest storage overhead.

2.5 Multimedia Caches

Multimedia caches have traditionally been used by content providers to cache contents near end-users in order to reduce bandwidth cost as well as access latency [55, 56]. The use of caches to improve the performance of multimedia distribution systems has been studied extensively in the past two decades. Tang et al. [98] provide a general overview on existing multimedia caching techniques. Wu et al. [101] were among the first to propose the caching of chunks (segments) of a video. Video chunking allows to cache popular chunks and prefetch first chunks to reduce initial buffering time of videos [92]. An important aspect of a good cache design is to decide when to store a data element in the cache and when to evict an existing data element to store new data elements. LRU [57] is a widely-used cache eviction policy that evicts the least recently used data element to store new elements. Another popular policy is LFU [58] that evicts the least frequently used data element, instead of the least recently used, to make room for new data elements.

In recent years, mobile operators have also started using multimedia caches to augment cell towers. The growth of smartphones as a primary end-point for multimedia data has led to a significant rise in bandwidth usage of cell towers, which requires higher backhaul bandwidth to fetch data. Traditionally, network operators have deployed caches only at centralized locations, such as operator peering points, in part, for both simplicity and ease of management [102]. However, researchers and startups have recently recognized the benefit of placing caches closer to edge [102, 84]. The effectiveness of caching for YouTube videos has also been studied in the past [87, 103]. Both investigations show that caching of YouTube video can both, on a global and regional level, reduce server and network load significantly.

Although multimedia caching has been thoroughly studied in the past two decades, there has been little work on designing caches for intermittent power. Similar to other caches, a multimedia cache is primarily used for performance optimization, and not for application correctness. So, unlike DFSs, multimedia clients would not stall when cached states become inaccessible at low power, but the benefit of caching becomes insignificant if the cache is not designed to handle intermittent power constraints. In Chapter 7 I propose several design techniques to avoid backhaul traffic for cached contents while minimizing buffering time at all power levels.
CHAPTER 3
WEATHER FORECASTS BASED ENERGY HARVESTING PREDICTION MODELS

To sustain perpetual operation, systems that harvest environmental energy must carefully regulate their usage to satisfy their demand. Regulating energy usage is challenging if a system’s demands are not elastic and its hardware components are not energy-proportional, since it cannot precisely scale its usage to match its supply. Instead, the system must choose when to satisfy its demands based on its current energy reserves and predictions of its future energy supply. In this chapter I propose the use of weather forecasts to improve a system’s ability to satisfy demand by improving its predictions.

3.1 Background and Motivation

Energy harvesting systems collect and store environmental energy to sustain continuous operation without access to external power sources. Energy-neutral systems always consume less than or equal to the energy they harvest [9]. An underlying goal of most energy harvesting systems is to operate as close to energy-neutral as possible to prevent downtime from battery depletions.

The strategy a system uses to achieve energy-neutral operation depends on the specific characteristics of its energy source, battery, hardware components, and workload. Achieving energy-neutral operation is simple if an energy source produces power faster than a system can consume it. Unfortunately, environmental energy sources, such as solar and wind, are intermittent and vary significantly over time due to weather conditions. As a result, these energy sources typically do not produce enough power to continuously operate a system’s hardware components. Instead, systems must adapt their energy usage over time to ensure they do not consume more energy than they are able to harvest and store.

Ideal hardware components are energy-proportional, such that their energy consumption scales linearly with their workload’s intensity [3]. Thus, a system with elastic workload demands achieves energy-neutral operation by changing the intensity of its workload, and hence its energy usage, at fine time-scales to match the energy it harvests. Prior work on energy harvesting primarily focuses on systems with energy-proportional components that have elastic workload demands [6, 8, 9, 11, 12, 19, 20, 21]. Maintaining energy-neutral operation in a system with inelastic workload demands using components that are not energy-proportional poses new challenges, since the system is unable to precisely change the intensity of its workload and energy usage to match the energy it harvests.

Instead, the system must choose how to satisfy its workload’s demands based on its current and expected energy supply. As others have noted, workload scheduling algorithms
in energy harvesting systems with inelastic demands are highly sensitive to energy harvesting predictions [14]. While past work recognizes the need for accurate energy harvesting predictions, prior prediction methods derive from the underlying idea that the past is an accurate predictor of the future [44, 10, 14, 15]. While the past is accurate for both sufficiently short, i.e., seconds to minutes, and sufficiently long, i.e., months to years, time-scales, we show in the next section that predictions derived from weather forecasts are more accurate at the medium-length time-scales, i.e., hours to days, relevant to a large class of energy harvesting systems. Our empirical findings match the same intuition that causes people to tune into a nightly weather forecast, rather than step outside, to find out the expected weather for the next few days. Our hypothesis is that energy harvesting predictions derived from weather forecasts for large regions improve nearby systems’ ability to satisfy their demand over the time-scales of hours to days, when compared against predictions derived from the immediate past.

In evaluating our hypothesis this chapter makes the following contributions.

**Analyze Historical Weather Data.** We analyze extensive traces of past forecast and observational weather data from the National Weather Service (NWS), as well as fine-grain solar and wind energy harvesting and observational weather data from our own deployment. We use these traces to quantify how well both weather forecasts and the immediate past predict the weather phenomena—sky condition and wind speed—that most impact solar and wind energy harvesting at time-scales ranging from 3 hours to 72 hours. We find that NWS forecasts in the regions we examine are a better predictor of the future than the immediate past at these time-scales for both sky condition and wind speed.

**Formulate Forecast→Energy Model.** We use our observational data to correlate (i) weather forecasts for our entire region with our own local weather observations and (ii) our own local weather observations with the energy harvested by our deployed solar panel and wind turbine. We use both data sets to formulate a simple model that predicts how much energy our solar panel and wind turbine will harvest in the future given weather forecasts every 3 hours from 3 hours to 72 hours in the future.

---

**Figure 3.1.** Power generated during a 12 day period in October, 2009 from our solar panel (a) and wind turbine (b).
Figure 3.2. The error in sky condition (a) and wind speed (b) when using the past to predict the future for different time intervals in 2008 at 1 hour and 5 minute granularities, respectively, for Amherst, Massachusetts.

3.2 The Case for Using Forecasts

To motivate the use of weather forecasts for prediction, we analyze both forecast and observational data from the year 2008 to compare the accuracy, at different time-scales, of predictions based on NWS forecasts with predictions based on the past. Others have noted that over appropriate time-scales and under ideal conditions the past predicts the future for both solar [2, 4, 9] and wind [10] power. However, our analysis leads to four observations that motivate the use of forecasts, instead of the past, for predictions over time-scales of hours to days. We use data from an extended deployment of a weather station, wind turbine, and solar panel on the roof of the Computer Science Building at the University of Massachusetts at Amherst, as well as data from NWS observations and the National Digital Forecast Database. Our observational traces are available from http://traces.cs.umass.edu and the NWS traces are available upon request from http://www.nws.noaa.gov/ndfd/.

Our weather station reports wind speed and solar radiation at 5 minute granularities, while the NWS reports an observation every hour and a forecast every 3 hours for every region of the country for the last 4 years. Each NWS forecast includes predictions every 3 hours from 3 hours to 72 hours in the future. Unless otherwise noted, we use our own weather station’s observations for Amherst, Massachusetts, and NWS observations for other regions. While our weather station and the NWS report a variety of weather metrics, we focus on the two metrics with the most direct relationship to the energy our solar panel and wind turbine harvest: sky condition, as a percentage of cloud cover between 0% and 100%, and wind speed, in units of miles per hour. We show how these metrics impact solar and wind energy harvesting in Section 3.3.

While we have found multiple approaches in prior work that use the immediate past to predict the future, the basic approach, which we term past predicts the future or PPF, we compare against predicts that a weather metric’s value in the next $N$ time units will exactly match the observations of that metric from the last $N$ time units. We discuss variants of this basic approach for solar power prediction in Section 3.5 that adapt to seasonal variations in
sunlight [9, 10, 19] or sudden changes in cloud cover [15]. We have found no prior work that focuses on variants of the PPF model for wind speed predictions.

The accuracy of the PPF model is dependent on the climate at a specific location. For example, a PPF model for solar power may be more accurate in areas with consistent sunlight and little variation in weather patterns, such as the desert in Australia [4], while a PPF model for wind power may be more accurate in areas likely to be in the path of a jet stream. Regardless of the area, though, prediction strategies without the aid of detailed weather forecasts must inherently rely on the past. Both our intuition and our empirical measurements lead to our first observation: there are many areas, including Amherst, Massachusetts, that do not have consistent weather patterns.

Observation #1: Both sky condition and wind speed show significant inter-day and intra-day variations, as a result of changing weather conditions in Amherst, Massachusetts, as well as other regions we examine, including Arizona, Florida, Washington, and Nebraska.

While we expect wind speeds to be intermittent, the data for the regions we examine also shows significant variations in the sky condition observed by the NWS both within each day and between days. As an example from our own deployment, Figures 3.1(a) and 3.1(b) show the solar and wind power we harvest, respectively, during a 12 day period in October, 2009. As expected, wind power is highly variable, with the wind turbine harvesting the most energy on days 3, 4, and 7, while harvesting lesser amounts on days 1, 6, 9, 10, and 12. The turbine harvests nearly zero energy on days 2, 5, 8, and 11. Surprisingly, despite its diurnal nature, solar power shows significant variations as well due to cloud cover, with the solar panel harvesting less than half its maximum possible energy on days 2, 3, 7, 8, and 11, with significant variations within each day. Our solar panel actually harvests no energy on day 11.

Even when the solar panel or wind turbine harvest the same amount of aggregate energy on two different days, the profile of power generation within each day is variable. For example, on both day 3 and 4 our solar panel harvests similar amounts of energy, but the power profile for day 4 is more consistent and less variable than day 3. Overall, the solar panel and wind turbine harvest less than $\frac{1}{2}$ their rated daily maximum on 40% and 75% of the days, respectively. While we chose a 12 day period to enhance the readability of the graph, we have witnessed a similar degree of day-to-day variation since the beginning of our solar panel and wind turbine deployment 4 months ago.

Observation #2: Using PPF to predict the future is least accurate at medium-length time-scales ranging from 3 hours to 1 week.

To evaluate the accuracy of the PPF model we focus on Amherst, Massachusetts, and calculate the root mean squared error (RMSE) between the average value of both sky condition and wind speed over an interval from $t = 0$ to $t = N$ and from $t = N$ to $t = 2N$ for all possible intervals of length $2N$ in the year 2008, given that our observational data has a granularity of 5 minutes. RMSE is a standard statistical measure of the accuracy of values predicted by a model with respect to the values observed. Intuitively, the value of the
RMSE quantifies the accuracy of the PPF model at different time-scales. For instance, an RMSE of zero for an interval of length $N$ indicates that for all possible intervals of length $N$ during the year the average of the metric in the previous interval exactly predicts the average of the metric in the next interval. The closer the RMSE is to zero for a particular interval duration the more accurate the past predicts the future for that interval.

Figures 3.2(a) and 3.2(b) show the RMSE for sky condition and wind speed, respectively, as a function of time interval duration $N$ ranging from 5 minutes to 6 months. Notice that we plot both graphs on a log scale. The analysis shows that predictions based on the past are most accurate at both short ($< 2$ minutes) and long time-scales ($> 10$ days), and are least accurate in between. For both sky condition and wind speed, the maximum inaccuracy occurs between 3 hours and one week, as indicated by each graph’s vertical lines.

**Observation #3:** Over the forecast time-scales from 3 hours to 3 days provided by the NWS, sky condition and wind speed forecasts are better predictors of the future than the PPF model.

We next show that NWS forecasts for the medium-length time-scales of hours to days are more accurate than the PPF model. To quantify the relative accuracy of weather forecasts, we use NWS forecast data from three months in different seasons—January, April, and September 2008—for Chicopee Falls, Massachusetts. Chicopee Falls, at 20 miles away, is the closest NWS site to Amherst. We first compare the accuracy of a forecast for sky condition with the accuracy of the PPF model from Figure 3.2(a). Figure 3.3 shows the RMSE between the observational sky condition and the sky condition from the NWS forecasts, as a function of the forecast time horizon. Since our weather station does not report sky condition, we use the hourly NWS observations of sky condition at Chicopee Falls, Massachusetts. As expected, the accuracy of the sky condition forecast decreases as the time horizon increases. Since the RMSE of the sky condition forecast ($< 30$) is less than the RMSE of the PPF model from Figure 3.2(a) between 3 hours and 3 days ($\sim 60$) we conclude that the forecast is a better predictor than the past for sky condition in Amherst, Massachusetts.

We next compare the accuracy of the NWS forecast for wind speed with the accuracy of the PPF model from Figure 3.2(b). Figure 3.3 shows the RMSE between the observational wind speed and the wind speed from the NWS forecast, as a function of the forecast time horizon. As the figure shows, the accuracy of the wind speed forecast does not vary significantly for any future time horizon. The cumulative distribution of errors in the wind speed forecast echoes the point, since the error for each time horizon is roughly equivalent, with 80% of the errors being less than 7mph. Since our wind turbine only generates power at wind speeds greater than 7 mph, we also examined the wind speed forecast accuracy after filtering out lower wind speeds. Our analysis shows that the accuracy of the wind speed forecast only increases as we raise the filtering threshold. Since the RMSE of the NWS wind speed forecast ($< 6$) is less than the RMSE of the PPF model from Figure 3.2(b) between 3 hours and 3 days ($\sim 11$), we conclude that the NWS forecast is a better predictor than the past for wind speed in Amherst, Massachusetts, which leads to our final observation.
Figure 3.3. RMSE between the observed sky condition and wind speed and those predicted by NWS forecasts from 3 hours to 72 hours in the future.

Figure 3.4. Relationship between the solar radiation our weather station observes and the power generated by our solar panel.

Observation #4: We conclude that using weather forecasts as a basis for prediction should be able to improve the performance of energy harvesting systems with inelastic demands that make workload decisions over 3 hour to 3 day time horizons.

3.3 Forecast → Energy Model

To leverage our observations from the previous section, we now formulate models that predict the power our solar panel and wind turbine will harvest given a NWS weather forecast. Note that our models are based on our specific solar panel and wind turbine, as well as the weather forecasts at our location. Since we derive our model parameters empirically, they depend on the specific characteristics of our deployment, and are not directly useful for other deployments. While the methods we use for building our models
may be applicable to other deployments, the accuracy we report is dependent on the specific characteristics of our location’s climate. Further, since we deploy our harvesting equipment in an open area, we do not evaluate the effect of local conditions, such as shade from foliage or wind shear from surrounding buildings, on our model.

Before discussing our model, we briefly describe our energy harvesting deployment, which consists of a battery, solar panel, and wind turbine. Air-X manufactures our wind turbine, and rates its maximum power output as 400 watts in 28 mile per hour winds. The turbine uses an internal regulator to govern the power delivered to the battery to prevent overcharging when the battery voltage increases beyond a threshold of 14.1 volts. Kyocera manufactures our solar panel, and rates its maximum power output as 65 watts at 17.4 volts under full sunlight. We connect the solar panel to a deep-cycle battery through a TriStar T-60 charge controller, which protects the battery from overcharging. Our battery has an ideal capacity of 1260 watt-hours.

To prevent our system’s battery from becoming fully charged, we use an additional T-60 load controller in conjunction with a 60 watt automotive bulb to bleed the battery’s energy. The controller connects the load to the battery at 13.6 volts and disconnects at 12.1 volts to ensure the battery stays charged to 55% of its capacity. The final component of our measurement system is a HOBO U30 wireless data logger. The logger measures battery voltage, using a built-in analog-to-digital converter, and electrical current, using an external current transducer for each energy source. The logger measures each quantity every 30 seconds and stores a 5 minute average locally. Each hour, the logger uploads its log file to a server hosted by HOBO, where data is publicly available for viewing through the HOBO web interface.

3.3.1 Sky Condition → Solar Power Model

We base our model for solar energy on a simple premise: if the sky condition reports a cloud cover of $N\%$ then the observed solar radiation, as well as our solar panel’s power production, will be $(100 - N)\%$ of the maximum possible under ideal cloudless skies. For example, if the 3 hour forecast predicts a sky condition with 50% cloud cover, and the maximum possible solar power production is 60 watts over that 3 hour interval, then the solar power prediction for that 3 hour interval will be $60 \times 0.5 = 30$ watts. Given our simple premise, to formulate our model, we must first estimate the maximum possible solar power production at any time of the day and year, given the tilt of the earth’s axis and the sun’s diurnal nature. Since our solar panel deployment has not been active for an entire year, we use our weather station’s traces of solar radiation to construct our model.

3.3.1.1 Computing Solar Power From Solar Radiation

We first derive the relationship between the solar radiation our weather station observes and the power our solar panel produces using our trace data, as shown in Figure 3.4. The relationship should be linear, since our solar panel produces energy in proportion to the solar radiation with a constant factor loss due to inefficiency. As expected, the relationship we observe is close to linear. We use the least-squares approach to fit the following regression line to the data, which we use to covert the solar radiation our weather station observes
Figure 3.5. Profile for solar power harvested on clear and sunny days in January, May, and September, and the quadratic functions \( f(x) \), \( g(x) \), and \( h(x) \) we fit to each profile, respectively.

to the solar power our panel produces, where power is in units of watts and solar radiation is in units of watt/m\(^2\).

\[
\text{SolarPower} = 0.0444 \times \text{Radiation} - 2.65
\]  
(3.1)

### 3.3.1.2 Computing the Maximum Possible Solar Power

We next derive our estimate for the maximum solar power possible at a given time of the day and year. The value is dependent on multiple factors, including the time of the day, day of the month, month of the year, and geographic location. While highly accurate models that take into account all of these factors are possible, we use a simple approximation that assumes the change in position of the sun relative to a specific location does not vary significantly within any single month. Thus, we use a profile for a single sunny day in each month of the year as the baseline for computing the ideal maximum power on any day of that month. We select a single sunny day with no cloud cover for each month from the year 2008 using our weather station data.

Figure 3.5 shows the profile of solar power our panel would harvest on three perfectly clear and sunny days in January 2008, May 2008, and September 2008. For the graph, we convert the solar radiation observed by our weather station on these days to the expected solar power harvested by our solar panel using equation (1) from above. We find that power appears quadratically related to the time of day. Since daylight hours change throughout the year, the power profile for a sunny day also changes. Of the three months, May has the maximum possible potential for power generation since it is nearest to the summer solstice, while January has the least possible potential for power generation since it is nearest to the winter solstice. For each month, we fit the quadratic function below, where \( a \), \( b \), and \( c \) for each month are given in Table 3.1, and \( \text{Time} \) is in hours after 12am.

\[
\text{MaxPower} = a \times (\text{Time} + b)^2 + c
\]  
(3.2)
<table>
<thead>
<tr>
<th>Month</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>January</td>
<td>-1.15</td>
<td>-12.75</td>
<td>21.45</td>
</tr>
<tr>
<td>February</td>
<td>-1.15</td>
<td>-12.75</td>
<td>29.13</td>
</tr>
<tr>
<td>March</td>
<td>-1.15</td>
<td>-12.75</td>
<td>35.97</td>
</tr>
<tr>
<td>April</td>
<td>-1.25</td>
<td>-13.5</td>
<td>43.72</td>
</tr>
<tr>
<td>May</td>
<td>-1.1</td>
<td>-13.5</td>
<td>43.5</td>
</tr>
<tr>
<td>June</td>
<td>-1.1</td>
<td>-13.5</td>
<td>43.4</td>
</tr>
<tr>
<td>July</td>
<td>-1</td>
<td>-13.5</td>
<td>40.35</td>
</tr>
<tr>
<td>August</td>
<td>-1.15</td>
<td>-13.5</td>
<td>40</td>
</tr>
<tr>
<td>September</td>
<td>-1.15</td>
<td>-13.5</td>
<td>36.32</td>
</tr>
<tr>
<td>October</td>
<td>-1</td>
<td>-13.35</td>
<td>27</td>
</tr>
<tr>
<td>November</td>
<td>-1.45</td>
<td>-12</td>
<td>22.66</td>
</tr>
<tr>
<td>December</td>
<td>-1.15</td>
<td>-12.5</td>
<td>16.79</td>
</tr>
</tbody>
</table>

Table 3.1. Values for a, b, and c in our quadratic solar power model, which is a function of the time of day for each month of the year.

### 3.3.1.3 Solar Model

To complete our model, we compute the solar power our panel generates using the equation below, where $MaxPower$ is in units of watts from equation (2) and $SkyCondition$ is the percentage cloud cover from the NWS. Figure 3.7(a) compares the observed solar power generated by our panel with the solar power predicted by our model. The graph demonstrates that the daily average difference between each observed and predicted value is small. Further, the model tends to be conservative when incorrect: the predictions are generally less than the observations, which reduces battery depletions from incorrect predictions.

$$
Power = MaxPower \times (1 - SkyCondition)
$$

### 3.3.2 Wind Speed $\rightarrow$ Wind Power Model

Our wind power model is simpler than our solar model, because, as opposed to sky condition, both our weather station and the NWS forecast report wind speed. Figure 3.6 shows the recorded power output of the wind turbine for different recorded wind speeds, as well as curves showing the power ratings for the turbine in both turbulent and steady winds. Wind power production is known to be a cubic function of the wind speed [1].

$$
Power = 0.01787485 \times (WindSpeed)^3 - 3.4013
$$
Figure 3.6. Power output from our wind turbine and the power output predicted by our wind power model over the first 3 weeks of October. The graph shows the rated power curves from the wind turbines manual for steady and turbulent wind, as well as our fitted curve.

We have developed a more accurate prediction. Note that the wind turbine stops producing power near 28 miles per hour, so our function ramps down to 0 at that point.

3.3.3 Compensating for Forecast Errors

Our solar and wind power models convert an observed sky condition and wind speed to the expected solar and wind power generated by our deployment. To convert a forecast for sky condition and wind speed to a prediction for solar and wind power we multiply the output of both models with an error constant $\alpha$. We base our $\alpha$ constant for each forecast time horizon on the RMSE for sky condition and wind speed forecasts in the previous section. Thus, the greater the expected error in the forecast at a particular future time, the smaller the value of $\alpha$ in our model.

3.4 Comparison with PPF Variants

In addition to the basic PPF model, we evaluate three PPF variants from prior work: Exponentially Weighted Moving Average (EWMA), Simple Moving Median (SMM), and Weather Conditioned Moving Average (WCMA). We briefly describe each approach.

EWMA divides a day into slots and predicts the energy for a particular slot as the weighted sum of the energy harvested in the same slot on N previous days [9]. EWMA assigns an exponentially decaying weighting factor to each previous day, since the recent past tends to provide more accurate predictions. We choose each slot’s length to be 60 minutes, since the environmental variation within each hour is typically small. Smaller time slots increase accuracy slightly at the expense of higher computational overhead. For EWMA, the predicted energy harvested at a time slot $t$ on $i$th day is given as:
Figure 3.7. Power output from our solar panel and the power output predicted by different prediction models over the first 3 weeks of October, 2009.

\[ E_{predicted}^t(i) = \alpha E_{observed}^t(i - 1) + (1 - \alpha) E_{predicted}^{t-1}(i) \]  

We empirically find that a weighting factor \( \alpha=0.1 \) minimizes the RMSE between the observed and predicted energy for our deployment based on historical data. SMM is an another variant of PPF, which predicts the energy for a particular slot as the median of energy harvested in the same slots on \( N \) previous days. SMM is more robust than EWMA to high fluctuations or other anomalies in the time series data.

Finally, WCMA is a variant of EWMA, which uses the current day’s, as well as previous days’, observational data to make predictions [17, 18]. In contrast to EWMA, WCMA considers the weather conditions of previous slots in the current day. As a result, it performs better than EWMA during inconsistent or fluctuating weather conditions. WCMA predicts energy for any time slot \( t \) on \( i^{th} \) day as

\[ E_{predicted}^t(i) = \alpha E_{t-1}^{observed}(i) + GAP_k(1 - \alpha) M_D^t(i) \]  

Where \( \alpha \) is a weighting factor similar to EWMA, and \( M_D^t(i) \) is the mean of the observed values in time slot \( t \) over the past \( D \) days. \( GAP_k \) is a factor that depends on past \( k \) slots and measures the present weather conditions compared to the same conditions over the previous days. Recas et al. [17] provide a detailed description of calculating \( GAP_k \). For WCMA, we find the optimal values of three parameters—\( \alpha, D, k \)—that minimize the...
Figure 3.8. Power output from our wind turbine and the power output predicted by different prediction models over the first 3 weeks of October, 2009.

RMSE between the observational energy and the predicted energy as $\alpha=0.4$, $D=6$, and $k=15$ for our solar panel, and $\alpha=0.9$, $D=7$, and $k=14$ for our wind turbine. Similar to EWMA, we assume a slot duration of 60 minutes. We use the optimal values for the EWMA and WCMA parameters in our evaluation.

Figure 3.7 compares the observed solar power generated by our panel with the solar power predicted by all four prediction models: (a) Forecast Predicts Future (FPF), (b) Exponentially Weighted Moving Average (EWMA), (c) Weather Conditioned Moving Average (WCMA), and (d) Simple Moving Median (SMM). Since the figure plots predictions only three hours into the future, it represents a best case scenario for the prediction models based on the past. The figure demonstrates that the forecast-based approach and WCMA perform significantly better than the EWMA or SMM model, especially when environmental conditions change. Although WCMA’s accuracy is similar on average to our forecast-based approach, it over-predicts on most days, which results in frequent battery depletions in energy harvesting systems.

Similarly, figure 3.8 compares the observed wind power generated by our turbine with the wind power predicted by all four prediction models. Again, WCMA and our forecast-based approach provide similar prediction accuracy for wind power, while EWMA and SMM are much less accurate due to the wind’s intermittent nature. WCMA also suffers from over prediction with wind energy, while our forecast-based approach tends to under predict when it is inaccurate.
3.5 Related Work

We know of no prior work that evaluates the use of forecast-based predictions in energy harvesting systems. Much of the prior work on energy harvesting systems assumes elastic workload demands that do not require predictions, since the system continually adapts its workload’s intensity and energy usage to match its energy supply [9, 19, 21]. However, while Moser et al. [14] assume perfect future knowledge of an energy source and do not investigate prediction strategies, they do note that scheduling algorithms for workloads with inelastic demands are highly sensitive to the accuracy of predictions. While our observation about the inter- and intra-day variations in solar radiation hold for Amherst, Massachusetts, prior work on solar harvesting assumes diurnal behavior that is more consistent than we observe [4, 23]. In these areas, the NWS forecast-based approach may be less effective.

Most prior work focuses on simple prediction schemes, such as the PPF model, based on the immediate past [9, 13]. As we show, the simple PPF approach is not as accurate as a NWS forecast-based approach for either solar or wind power at time-scales of hours to days. Kansal et al. [9] maintain an exponentially weighted moving average (EWMA) for solar power to achieve energy-neutral operation in a system with elastic workload demands. The EWMA approach is a variant of PPF that adapts to seasonal variations in solar radiation. However, EWMA does not account for drastic changes in weather that the NWS forecast predicts. Noh et al. [15] use a historical model for solar radiation that maintains an expectation for each time slot in a day based on the previous day’s solar radiation reading, but down-scales all future time-slots in a day by \( N\% \) if it records a solar radiation reading \( N\% \) less than expected. We plan to study how these variants of PPF compare with our forecast-based approach as part of future work.

The techniques above do not apply to wind speed or wind power predictions, since the wind is more intermittent than solar radiation and not diurnal in nature. We know of no work that discusses prediction strategies for wind speed. The recent commoditization and emergence of micro-wind turbines, such as the 400 watt Air-X we use in our deployment, motivates further study of harnessing wind power in energy harvesting systems deployed at locations with ample wind but little sunlight, i.e., during the winter in the extreme north or south.

3.6 Conclusion

In this chapter, we show how to leverage weather forecasts provided by the NWS to enhance the ability of energy harvesting systems to satisfy their demand. We analyze observational weather data from our own weather station, energy harvesting data from our own solar panel and wind turbine, and NWS observational and forecast data. Our analysis shows that weather predictions based on NWS forecasts are more accurate than predictions based on the past in many regions of the United States, including Amherst, Massachusetts. To leverage NWS forecasts in distributed systems, we formulate a model for our solar panel and wind turbine that converts the forecast to an energy harvesting prediction. We then compare our models with PPF approaches and show that our models outperform the PPF approaches at medium time scales from hours to days.

24
CHAPTER 4

MACHINE LEARNING MODEL FOR SOLAR ENERGY HARVESTING PREDICTION

The accuracy of an energy harvesting prediction model dictates the performance of distributed systems running on renewables. Since sky cover is not the only parameter that affects solar intensity, a simple empirical model, for predicting solar energy harvesting, based on sky cover forecast fails to account the impacts of other weather parameters on solar intensity. Furthermore, designing a simple empirical model based on all weather parameters is extremely challenging. To address the problem, in this chapter, I explore the use of powerful statistical techniques like machine learning techniques to automatically create site-specific prediction models for solar power generation based on the forecast of all weather parameters. I compare multiple regression techniques for generating prediction models, including linear least squares and support vector machines using multiple kernel functions.

4.1 Background and Motivation

A key goal of designing distributed systems running on renewables is to predict energy harvesting as accurately as possible. To facilitate better planning and optimize the performance of distributed systems, we focus on the problem of automatically generating models that accurately predict renewable generation using National Weather Service (NWS) weather forecasts. Specifically, we experiment with a variety of machine learning techniques to develop prediction models using historical NWS forecast data, and correlate them with generation data from solar panels. Once trained on historical forecast and generation data, our prediction models use NWS forecasts for a small region to predict future generation over several time horizons. Our experiments in this chapter use solar intensity as a proxy for solar generation, since it is proportional to solar power harvesting [94]. Importantly, since we generate our models from historical site-specific observational power generation data, they inherently incorporate the effects of local characteristics on each site’s capability to generate power, such as shade from surrounding trees. Since local characteristics influence power generation, individual sites must tune prediction models for site-specific characteristics.

Our goal is to automate generating prediction models that can be used in a variety of areas including distributed systems, smart grids, and smart homes. Distributed systems can use these prediction models to balance demand and supply, and to optimize the performance at all power levels. Both the grid and individual smart homes may use these prediction
models for advance planning of electricity generation and consumption. The grid can use the models to plan generator dispatch schedules in advance as the fraction of renewables increases in the grid. Smart homes can use the models to potentially plan their consumption patterns to better match the power that they generate on-site. In all cases, better prediction models are a prerequisite for increasing efficiency and encouraging broader adoption of distributed generation from renewables in computing infrastructure, the grid, and at smart homes. In studying prediction models for solar energy harvesting, we make the following contributions.

- **Data Analysis.** We analyze extensive traces of historical data from a weather station, as well as the corresponding NWS weather forecasts, to correlate the weather metrics present in the forecast with the solar intensity, in watts per m$^2$, recorded by the weather station. Our analysis quantifies how each forecast parameter affects each other and the solar intensity. For solar energy harvesting, we find that sky cover, relative humidity, and precipitation are highly correlated with each other and with solar intensity, while temperature, dew point, and wind speed are only partially correlated with each other and with solar intensity.

- **Model Generation.** We apply multiple machine learning techniques to derive prediction models for solar intensity using multiple forecast metrics, and then analyze the prediction accuracy of each model. We use machine learning on a training data set of historical solar intensity observations and forecasts to derive a function that computes future solar intensity for a given time horizon from a set of forecasted weather metrics. We formulate models based on linear least squares regression, as well as support vector machines (SVM). We find that SVM with radial basis function kernels built using historical data from seven weather metrics is 27% more accurate than existing forecast-based models that use only sky condition for predictions [94] and is 51% better than simple approaches that only use the past to predict the future.

In Section 4.2 we analyze forecast metrics and explore how they affect each other, as well as how they affect solar intensity, while in Section 4.3 we describe and evaluate multiple machine learning strategies for generating prediction models using our weather station data and NWS forecasts. Finally, Section 4.4 concludes this chapter.

### 4.2 Data Analysis

We collect weather forecast data and observational solar intensity data for 10 months starting from January 2010. We obtain historical forecast data from the NWS, which we have been collecting for the past 2 years. The NWS provides historical textual forecasts for small city-size regions throughout the U.S., which include multiple weather metrics for every hour of every day for the last few years. Each forecast includes predictions of each metric every 1 hour from 1 hour to 6 days into the future. Examples of weather metrics include temperature, dew point, wind speed, sky cover, probability of precipitation, and relative humidity. Sky cover is an estimate of the percentage (0%-100%) of cloud coverage in the atmosphere. In addition to making historical forecasts available, the NWS also
operates a real-time web service that enables applications to retrieve forecasts programmatic-ally as they become available. In addition to these metrics, we include the specific day of the year and time of the day as metrics, since daylight influences solar intensity and varies throughout the year for a given location.

We use observational solar intensity data from an extended weather station deployment on the roof of the Computer Science Building at the University of Massachusetts Amherst. The weather station reports solar intensity in watts/m² every 5 minutes of every day. Traces from the weather station are available at http://traces.cs.umass.edu. As we show in previous work, power generation from solar panels is directly proportional to solar intensity [94]; in general, solar panel inefficiencies result in power output that is a fixed percentage decrease from the raw solar intensity readings at the same location. We use NWS forecasts for Amherst, Massachusetts. In this section, we study how solar intensity varies with individual forecast parameters and how these forecast parameters are related to each other. The purpose of our data analysis is to provide intuition into how solar intensity and solar panel power generation depends on a combination of multiple weather metrics, and is not easily
Figure 4.3. Solar intensity generally decreases with increasing values of sky cover (a), relative humidity (b), and precipitation potential (c).

predictable from a single weather metric. The complexity in predicting solar intensity from one or more weather metrics motivates our study of automatically generating prediction models using machine learning techniques in the next section.

Fig. 4.1 shows how the day of the year affects solar intensity by charting the average solar intensity reading at noon per day over our 10 month monitoring period, where day zero is January 1st, 2010. As expected, the graph shows that the solar intensity is lowest in January near the winter solstice and increases into the summer before decreasing after the vernal equinox. Additionally, the graph also implies that other conditions also have a significant impact on solar intensity, since many days throughout the spring and summer have low solar intensity readings. The graph shows that solar intensity and the day of the year are roughly correlated: most of the time, but not always, a summer day will have a higher solar intensity than a winter day. However, other factors must contribute to the solar intensity, since there are clearly some sunny winter days that record higher solar intensity readings than some cloudy summer days. To better understand correlations with other weather metrics, we model similar relationships for the other forecast metrics.

For example, Fig. 4.2 shows that wind speed, dew point, and temperature are not highly correlated with solar intensity. Solar intensity varies almost uniformly from lower to higher values at any value of wind speed (a). Thus, wind speed has nearly zero correlation with solar intensity and its value is not indicative of the solar intensity or solar panel power generation. Both temperature (b) and dew point (c) correlate with solar intensity at higher values: if the temperature or dew point is high, then the solar intensity is likely to be high. However, if the temperature or dew point is low, the solar intensity exhibits a more significant variation between high and low values. The results are intuitive. For example, in the summer a high temperature is often dependent on sunlight, while in the winter sunlight contributes less in raising the ambient temperature.

In contrast, Fig. 4.3 shows that sky cover, relative humidity, and chance of precipitation have high negative correlations with solar intensity. In each case, as the value of the metric increases, the solar intensity reading generally decreases. However, as with the day of the year, there must be other factors that contribute to the solar intensity reading, since there are some days with a high sky cover, relative humidity, and precipitation probability, but a high solar intensity reading and vice versa. In addition to exhibiting complex relationships with solar intensity, each weather metric also exhibits a complex relationship with other weather
metrics. For example, Fig. 4.4 shows that relative humidity (a) and chance of precipitation (b) exhibit strong, but not perfect correlations, with sky cover, while relative humidity is strongly correlated with chance of precipitation (c). In all three cases, the metrics rise in tandem, although the relationship is noisy due to the value of other weather metrics.

Table 1 shows correlation coefficients for each weather metric using the Pearson product-moment correlation coefficient, which divides the covariance of the two variables by the product of their standard deviations. The higher the absolute value of the correlation coefficient, the stronger the correlation between the two weather metrics—a positive correlation indicates an increasing linear relationship, while a negative correlation indicates a decreasing linear relationship. The complex relationships between weather metrics and solar intensity shown in this table motivate our study of automated prediction models using machine learning techniques in the next section.

### 4.3 Prediction Models

We represent both observational and forecast weather metrics as a time-series that changes due to changing weather patterns and seasons. As the previous section shows, solar intensity depends on multiple weather metrics, which complicates the task of developing an accurate prediction model. The high dimensionality of the time-series data motivates our study of regression methods to develop solar intensity prediction models.
To generate each model we provide eight months of training data (January to August) as input, which includes solar intensity readings as well as NWS forecasts for 6 weather metrics. The machine learning techniques automatically output a function that computes solar intensity from the 6 weather metrics, as well as the day of the year. We use the remaining 2 months of our data set to test the model’s accuracy. One benefit of using machine learning to automatically generate prediction models is that, in general, the more training data that is available, the more accurate the model.

We focus our study on short-term forecasts three hours in the future. For our experiments, we develop models that determine a relationship at any time $t$ between the solar intensity and the forecast weather metrics three hours in the past ($t - 3$). Note that we are able to apply our techniques to forecasts of any length; we choose three hours as a simple illustration. Using our models and the three hour forecast, we are able to compute a prediction for the solar intensity three hours in the future. The models we generate are simple functions, of the form below, that compute solar intensity from multiple weather metrics including the day of the year. We could also add time of the day as an additional metric, but for ease of exposition our experiments focus on predictions at noon. We compare the accuracy of our models with each other, as well as against a simple model we developed in prior work [94] based solely on the sky condition metric and against a simple past-predictsfuture model. Our previous model multiplies the maximum power a solar panel is able to generate at a given time (of the day and year) by (1-SkyCover), since sky cover represents an estimate of the percentage of the atmosphere the sun is covering.

\[
\text{SolarIntensity} = F(\text{Day}, \text{Temperature}, \text{DewPoint}, \text{WindSpeed}, \text{SkyCover}, \text{Precipitation}, \text{Humidity})
\]

$F$ is the function that we determine using different regression methods. We preserve the units of each metric: we represent each day as a value between 0 and 365, temperature in degrees Fahrenheit, wind speed in miles per hour, sky cover in percentage between 0% and 100%, precipitation potential in percentage between 0% and 100%, and humidity in percentage between 0% and 100%. However, before applying any regression techniques below we normalize all feature data to have zero mean and unit variance. To quantify the accuracy of each model, we use the Root Mean Squared Error (RMS-Error) between our predicted solar intensity at any time and the actual solar intensity observed. RMS-Error is a well-known statistical measure of the accuracy of values predicted by a time-series model with respect to the observed values. An RMS-Error of zero indicates that the model exactly predicts solar intensity three hours in the future. The closer the RMS-Error is to zero the more accurate the model’s predictions.

### 4.3.1 Linear Least Squares Regression

We first apply a linear least squares regression method to predict solar intensity. Linear least squares regression is a simple and commonly-used technique to estimate the relationship between a dependent or response variable, e.g., solar intensity, and a set of independent variables or predictors. The regression minimizes the sum of the squared differences between the observed solar intensity and the solar intensity predicted by a linear approximation of the forecast weather metrics. Applying the linear least squares method to the
Figure 4.5. Observed and predicted solar intensity using linear least squares regression for September and October 2010.

eight months of training data yields the prediction model below, with coefficients for each metric.

\[
\text{SolarIntensity} = 1.18 \times \text{Day} + 77.9 \times \text{Temp} + 33.11 \times \text{DewPoint} + 22.8 \times \text{WindSpeed} - 96.9 \times \text{SkyCover} - 49.15 \times \text{Precipitation} - 43.4 \times \text{Humidity}
\]

We verify the prediction accuracy using our test dataset for the remaining months of the year. We observe the cross validation RMS-Error and prediction RMS-Error in the solar intensity as 165 watts/m\(^2\) and 130 watts/m\(^2\), respectively. We cross validate the regression model with the training dataset (from Jan. to Aug. 2010) and verify its prediction accuracy using the testing dataset (Sept. and Oct. 2010). The cross validation RMS-Error quantifies how well the model predicts values in the training data set, while the prediction RMS-Error predicts how well the model predicts values in the testing data set. Fig. 4.5 shows the observed and predicted solar intensity for September and October 2010. As the figure shows, the model tracks the solar intensity prediction reasonably accurately, albeit with a few deviations.

4.3.2 Support Vector Machines

We next look at multiple classes of supervised learning methods using Support Vector Machines (SVM) [27]. SVMs, which construct hyperplanes in a multidimensional space, have recently gained popularity for classification and regression analysis. The accuracy of SVM regression depends on the selection of an appropriate kernel function and parameters. In our work, we studied three distinct SVM kernel functions: a Linear Kernel, a Polynomial Kernel, and a Radial Basis Function (RBF) kernel. An SVM uses the kernel function to transform data from the input space to the high-dimensional feature space. We chose SVMs over other supervised learning methods due to its sparsity property and its ability to handle non-linearity in the data. We use the LibSVM library, which includes a multitude of SVM regression techniques, to implement SVM regression with the linear kernel function on our training data set [28]. We found that both the linear and polynomial kernel performed poorly with RMS-Errors of 201 watts/m\(^2\) and 228 watts/m\(^2\), both of which were worse
Figure 4.6. Observed and predicted solar intensity, using SVM regression with an RBF kernel, for the months of September and October 2010.

Figure 4.7. Observed and predicted solar intensity, using three different prediction techniques — (a) SVM-RBF kernel with 4 dimensions, (b) cloudy computing model using sky condition forecast, (c) past predicts future prediction model — for the months of September and October 2010.

than the linear least squares approach above. As a result, we focus on results using the RBF kernel.

We tested the RBF kernel and SVM using the LibSVM library on our eight months of training data. In order to find the optimal parameters for the RBF kernel we ran a grid search tool from the LibSVM library on the training dataset. We found the optimal parameters of the RBF kernel to be $\text{cost} = 256$, $\gamma = 0.015625$, and $\epsilon = 0.001953125$. Using these parameters, we found that the RBF kernel using all seven dimensions for the seven weather metrics provides a better regression model than the other regression methods, as indicated by its low cross validation and prediction errors, both of which are significantly lower than either the linear or the polynomial kernels. Fig. 4.6 also reflects this fact in the time-series graph of observed and predicted values. The results show that the RBF kernel also performs better than the linear least squares method, having a slightly lower cross validation RMS-error (164 watts/m$^2$) and a slightly higher prediction RMS-Error (163 watts/m$^2$).
4.3.3 Eliminating Redundant Information

As we show in the previous section, many weather metrics show a strong correlation with each other. As a result, our SVM regression models contain redundant information, which often decreases the prediction accuracy of each model. Principal component analysis (PCA) is a popular method for removing redundant informations from an input dataset, thereby reducing its dimensionality [29]. Thus, we use the principal component analysis algorithm to remove redundant informations from our feature dataset. The PCA algorithm uses an orthogonal transformation to convert a set of, potentially correlated, input variables into a set of uncorrelated variables called principal components. The number of principal components is less than or equal to the number of original variables. The first principal component has the maximum possible variance, and the second principal component has the maximum possible variance under the constraint that it be orthogonal to the first component, etc.

We choose the first four principal components corresponding to first four (highest) eigenvalues and run the RBF SVM regression method on the reduced feature-set. The results (Fig. 4.7(a)) show that the RBF kernel performs better after PCA analysis than when using the full feature-set with a cross validation RMS-Error of 159 watts/m$^2$ and a prediction RMS-error of 128 watts/m$^2$, both of which outperform the linear least squares model. We also ran experiments that further reduced the dimensionality of the feature set from 4 to 2. However, we found that all three SVM regression techniques performed worse compared to the 4-dimentional feature set. The performance degradation is the result of the additional reduction in dimensionality eliminating information that aids in prediction and is not redundant.

4.3.4 Comparing with Existing Models

Finally, we compare our regression-based prediction models with existing models. First, we compare with a past-predicts-future model (PPF), which uses the previous day solar intensity to predict the next day solar intensity. Past predicts future models are typically used when forecasts are not available, since the past is a reasonably good indicator of the future if the weather does not change. While they are highly accurate if weather conditions do not change, the models are not able to predict drastic changes in the weather. Second, we compare with a simple model that uses only the sky condition as a basis for prediction, called cloudy, which we developed in prior work [94]. We have shown that the cloudy model is more accurate than existing variants of PPF in the literature [30]. While the model is able to predict changes in weather, it does not incorporate information from multiple weather metrics and their impact on solar intensity.

Fig. 4.7(b) and (c) show how well cloudy and PPF predict weather in our testing data set, respectively. The results show that while the cloudy model follows the general trend of the weather it frequently exhibits wrong predictions. As expected, PPF’s results are inaccurate whenever weather changes, which happens nearly every day. By contrast, Fig. 4.7(a) shows that SVM-RBF with the reduced feature provides a much more accurate model. The RMS-Errors for each model highlight this result: the RMS-Error for SVM-RBF with four dimensions is 128 watts/m$^2$, while the RMS-Error for cloudy and PPF is 175 and 261, re-
spectively. Thus, SVM-RBF with four dimensions is 27% more accurate than the simple cloudy model and 51% more accurate than the PPF model.

4.4 Conclusion

Prior prediction models for solar energy harvesting have been based primarily on the immediate past [31, 32, 33]. Unfortunately, these methods are unable to predict changes in weather patterns in advance. Since weather forecasts from the NWS are based on aggregations of multiple data sources from across the country, they are able to provide advance warning. The NWS generates forecasts from multiple sophisticated forecast models that synthesize a multitude of observational data. We show that the relationship between these forecast weather metrics and solar intensity is complex. Thus, we automatically derive prediction models from historical solar intensity and forecast data using machine learning techniques. We find that models derived using SVMs with RBF kernels and linear least squares outperform a past-predicts-future models and a simple model based on sky condition forecasts from prior work [94] and is a promising area for increasing the accuracy of solar power generation prediction.
CHAPTER 5

MANAGING SERVER CLUSTERS ON INTERMITTENT POWER

The previous two chapters have described weather forecasts based empirical and machine learning models to predict future energy harvesting from renewables. This chapter presents a new energy abstraction to manage server clusters on renewables or intermittent power in general. In this chapter I propose blinking – metered transitions between a high-power active state and a low-power inactive state – as the primary abstraction for conforming to intermittent power constraints. I design Blink, an application-independent hardware-software platform for developing and evaluating blinking applications, and define multiple types of blinking policies. I then use Blink to design BlinkCache, a blinking version of Memcached, to demonstrate the effect of blinking on an example application.

5.1 Background and Motivation

Energy-related costs have become a significant fraction of total cost of ownership (TCO) in modern data centers. Recent estimates attribute 31% of TCO to both purchasing power and building and maintaining the power distribution and cooling infrastructure [112]. Consequently, techniques for reducing the energy footprint of data centers continue to receive significant attention in both industry and the research community. We categorize these techniques broadly as being either primarily workload-driven or power-driven. Workload-driven systems reconfigure applications as their workload demands vary to use the least possible amount of power to satisfy demand. Examples include consolidating load onto a small number of servers, e.g., using request redirection [49, 41] or VM migration, and powering down the remaining servers during off-peak hours, or balancing load to mitigate thermal hotspots and reduce cooling costs [37, 60, 61]. In contrast, power-driven systems reconfigure applications as their power supply varies to achieve the best performance possible given the power constraints.

While prior work has largely emphasized workload-driven systems, power-driven systems are becoming increasingly important. For instance, data centers are beginning to rely on intermittent renewable energy sources, such as solar and wind, to partially power their operations [47, 72]. Intermittent power constraints are also common in developing regions that experience “brownouts” where the electric grid temporarily reduces its supply under high load [41, 75]. Price-driven optimizations, due to either demand-response incentives or market-based pricing, introduce intermittent constraints as well, e.g., if multiple data centers coordinate to reduce power at locations with high spot prices and increase power at locations with low spot prices [66]. Variable pricing is an important tool for demand-side power management of future smart electric grids. The key challenge in power-driven
systems is optimizing application performance in the presence of power constraints that may vary significantly and frequently over time. Importantly, these power and resource consumption constraints are independent of workload demands.

In this chapter, we present Blink, a new energy abstraction for gracefully handling intermittent power constraints. Blinking applies a duty cycle to servers that controls the fraction of time they are in the active state, e.g., by activating and deactivating them in succession, to gracefully vary their energy footprint. For example, a system that blinks every 30 seconds, i.e., is on for 30 seconds and then off for 30 seconds, consumes half the energy, modulo overheads, of an always-on system. Blinking generalizes the extremes of either keeping a server active (a 100% duty cycle) or inactive (a 0% duty cycle) by providing a spectrum of intermediate possibilities. Blinking builds on prior work in energy-aware design. First, several studies have shown that turning a server off when not in use is the most effective method for saving energy in server clusters. Second, blinking extends the PowerNap [54] concept, which advocates frequent transitions to a low-power sleep state, as an effective means of reducing idle power waste.

An application’s blinking policy decides when each node is active or inactive at any instant based on both its workload characteristics and energy constraints. Clearly, blinking impacts application performance, since there may not always be enough energy to power the nodes necessary to meet demand. Hence, the goal of a blinking policy is to minimize performance degradation as power varies. In general, application modifications are necessary to adapt traditional server-based applications for blinking, since these applications implicitly assume always-on, or mostly-on, servers. Blinking forces them to handle regular disconnections more often associated with weakly connected [73] environments, e.g., mobile, where nodes are unreachable whenever they are off or out of range.

5.1.1 Example: BlinkCache

To demonstrate how blinking impacts a common data center application, we explore the design of BlinkCache—a blinking version of memcached that gracefully handles intermittent power constraints—as a proof-of-concept example. Memcached is a distributed memory cache for storing key-value pairs that many prominent Internet sites, including LiveJournal, Facebook, Flickr, Twitter, YouTube, and others, use to improve their performance.

Memcached is a natural first application to optimize for variable power constraints for two reasons. First, a memcached cluster is energy-intensive, since it requires continuous operation of high-memory nodes to ensure instant access to in-memory state. Second, since memcached is performance supplement that is not necessary for correctness, it does not preclude a constrained power source that may offer little or no power over some time periods. A blinking memcached cluster also exploits the increasing trend toward elevating memory in the storage hierarchy, and using it as the primary storage substrate in cloud data centers [63]. An important consequence of this trend is that applications increasingly use memory less like a cache that only stores a small set of popular objects, and more like a storage system that also stores a long heavy tail of unpopular objects. However, unpopular objects are not unimportant.
Figure 5.1. The popularity of web data often exhibits a long heavy tail of equally unpopular objects. This graph ranks the popularity of Facebook group pages by their number of fans.

For Internet services that store user-generated content, the typical user is often interested in the relatively unpopular objects in the heavy tail, since these objects represent either their personal content or the content of close friends and associates. As one example, Figure 5.1 depicts a popularity distribution for Facebook group pages in terms of their number of fans. While the figure only shows the popularity rank of the top 10,000 pages, Facebook has over 20 million group pages in total. Most of these pages are nearly equally unpopular. For these equally unpopular objects, blinking nodes synchronously to handle variable power constraints results in fairer access to the cache. While fair cache access is important, maximizing memcached’s hit rate requires prioritizing access to the most popular objects. We explore these performance tradeoffs in-depth for a memcached cluster with intermittent power constraints.

5.1.2 Contributions

In designing, implementing, and evaluating BlinkCache as a proof-of-concept example, this chapter makes the following contributions.

• Make the Case for Blinking Systems. We propose blinking systems to deal with variable power constraints in server clusters. We motivate why blinking is a beneficial abstraction for dealing with intermittent power constraints, define different types of blinking policies, and discuss its potential impact on a range of distributed applications.

• Design a Blinking Hardware/Software Platform. We design Blink, an application-independent hardware/software platform to develop and evaluate blinking applications. Our small-scale prototype uses a cluster of 10 low-power motherboards connected to a programmable power meter that replays custom power traces and variable power traces from a solar and wind energy harvesting deployment.

• Design, Implement, and Evaluate BlinkCache. We use Blink to experiment with blinking policies for BlinkCache, a variant of memcached we optimize for intermittent power constraints. Our hypothesis is that a load-proportional blinking policy,
which keeps nodes active in proportion to the popularity of the data they store, combined with object migration to group together objects with similar popularities, results in near optimal cache hit rates, as well as fairness for equally unpopular objects. To validate our hypothesis, we compare the performance of activation, synchronous, and load-proportional policies for realistic Zipf-like popularity distributions. We show that a load-proportional policy is significantly more fair than an optimal activation policy for equally popular objects (4X at low power) while achieving a comparable hit rate (over 60% at low power).

Section 5.2 provides an overview of blinking systems and potential blinking policies. Section 5.3 presents Blink’s hardware and software architecture in detail, while Section 5.4 presents design alternatives for BlinkCache, a blinking version of memcached. Section 5.5 then evaluates BlinkCache using our Blink prototype. Finally, Section 5.6 discusses related work and Section 5.7 concludes.

5.2 Blink: Rationale and Overview

Today’s computing systems are not energy-proportional [3]—a key factor that hinders data centers from effectively varying their power consumption by controlling their utilization. Designing energy-proportional systems is challenging, in part, since a variety of server components, including the CPU, memory, disk, motherboard, and power supply, now consume significant amounts of power. Thus, any power optimization that targets only a single component is not sufficient, since it reduces only a fraction of the total power consumption [3, 51]. As one example, due to the power consumption of non-CPU components, a modern server that uses dynamic voltage and frequency scaling in the CPU at low utilization may still operate at over 50% of its peak power [39, 74]. Thus, deactivating entire servers, including most of their components, remains the most effective technique for controlling energy consumption in server farms, especially at low power levels that necessitate operating servers well below 50% peak power on average.

However, data centers must be able to rapidly activate servers whenever workload demand increases. PowerNap [54] proposes to eliminate idle power waste and approximate an energy-proportional server by rapidly transitioning the entire server between a high-power active state and a low-power inactive state. PowerNap uses the ACPI S3 state, which places the CPU and peripheral devices in sleep mode but preserves DRAM memory state, to implement inactivity. Transition latencies at millisecond-scale, or even lower, may be possible between ACPI’s fully active S0 state and its S3 state. By using S3 to emulate the inactive “off” state, 1 PowerNap is able to consume minimal energy while sleeping. Typical high-end servers draw as much as 40x less power in S3.

Blink extends PowerNap in important ways. First, PowerNap is a workload-driven technique that eliminates idle server power waste—it uses rapid transitions in a workload-driven fashion to activate each server when work arrives and deactivate it when idle. In contrast, Blink is a power-driven technique that regulates average node power consumption

---

1We use “active” and “on” interchangeably to reference ACPI’s S0 state, and inactive and “off” interchangeably to represent ACPI’s S3 state.
### Table 5.1

<table>
<thead>
<tr>
<th>Type</th>
<th>Model</th>
<th>S3 Transition Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desktop</td>
<td>Optiplex 745</td>
<td>13.8</td>
</tr>
<tr>
<td>Desktop</td>
<td>Dimension 4600</td>
<td>12.0</td>
</tr>
<tr>
<td>Laptop</td>
<td>Lenovo X60</td>
<td>11.7</td>
</tr>
<tr>
<td>Laptop</td>
<td>Lenovo T60</td>
<td>9.7</td>
</tr>
<tr>
<td>Laptop</td>
<td>Toshiba M400</td>
<td>9.1</td>
</tr>
<tr>
<td>Laptop</td>
<td>OLPC-XO (w/ NIC)</td>
<td>1.6</td>
</tr>
<tr>
<td>Laptop</td>
<td>OLPC-XO (no NIC)</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Latencies for several desktop and laptop models to perform a complete S3 cycle (suspend and resume). Data from both [35] and our own measurements of Blink’s OLPC-XO.

The **blink state** of each node \( i \) is defined by two parameters that determine its duty cycle \( d_i \), (i) length of the ON interval \( t_{on} \) and (ii) length of the OFF interval \( t_{off} \), such that

\[
d_i = \frac{t_{on}}{t_{on} + t_{off}} \cdot 100\%
\]

A **blink policy** defines the blink state of each node in a cluster, as well as a blink schedule for each node.

The blink schedule defines the clock time at which a specified node transitions its blink state to active, which in turn dictates the time at which the node turns on and goes off. The schedule allows nodes to synchronize their blinking with one another, where appropriate. For example, if node \( A \) frequently accesses disk files stored on node \( B \), the blink policy should specify a schedule such that the nodes synchronize their active intervals. To illustrate how a data center employs blinking to regulate its aggregate energy usage, consider a scenario where the energy supply is initially plentiful and there is sufficient workload demand for all nodes. In this case, a feasible policy is to keep all nodes continuously on.

Next assume that the power supply drops by 10\%, and hence, the data center must reduce its aggregate energy use by 10\%. There are several blink policies that are able to satisfy this 10\% drop. In the simplest case, 10\% of the nodes are turned off, while the remaining nodes continue to stay on. Alternatively, another blink policy may specify a duty cycle of \( d_i = 90\% \) for every node \( i \). There are also many ways to achieve a per-server duty cycle of 90\% by setting different \( t_{on} \) and \( t_{off} \) intervals, e.g., \( t_{on} = 9s \) and \( t_{off} = 1s \) or \( t_{on} = 900ms \) and \( t_{off} = 100ms \). Yet another policy may assign different blink states

---

\(^2\)PowerNap’s on-demand transitions show little benefit once latencies exceed 100 milliseconds [54].
to different nodes, e.g., depending on their loads, such that aggregate usage decreases by 10%.

We refer to the first policy in our example above as an activation policy. An activation policy only varies the number of active servers at each power level [41, 74] such that some servers are active, while others are inactive; the energy supply dictates the size of the active server set. In contrast, synchronous policies toggle all nodes between the active and inactive state in tandem. In this case, all servers are active for $t_{on}$ seconds and then inactive for $t_{off}$ seconds, such that total power usage over each duty cycle matches the available power. Of course, since a synchronous policy toggles all servers to active at the same time, it does not reduce peak power, which has a significant impact on the cost of energy production. An asynchronous policy may randomize the start of each node’s active interval to decrease peak power without changing the average power consumption across all nodes. Finally, an asymmetric policy may blink different nodes at different rates, while ensuring the necessary change in the energy footprint. For example, an asymmetric policy may be load-proportional and choose per-node blink states that are a function of current load.

All of the policies above are equally effective at capping the average power consumption for a variable power signal over any time interval. However, the choice of the blink policy greatly impacts application performance. To see why, consider two common applications: a simple cluster-based web server [41, 49] and an Hadoop cluster [52]. An activation policy is well-suited for a cluster-based web server where each node serves static content replicated on each node—by turning off a subset of nodes and evenly redirecting incoming web requests to active servers, the application is able to regulate its energy usage to match its supply. Since any node is able to service any request, transitioning a subset of the nodes to the inactive state does not pose a problem. In this case, the application requires only minimal changes to accommodate dynamic changes in the set of active servers.

However, an activation policy presents a problem for applications that maintain memory or disk state on specific nodes and exhibit inter-node dependencies. In this case, deactivating nodes will render some application-specific state unavailable. Hadoop is one example, since Hadoop’s HDFS file system [68] replicates and stores data chunks of each file across many different nodes to improve both performance and data availability. As a result, simply powering down some nodes for long periods is not effective, since inactive nodes may store data necessary for a job to continue execution [52, 38, 48]. One, potentially non-optimal, option for addressing the problem without incurring the overhead of changing the data layout or migrating state prior to deactivating nodes is to leverage a synchronous blinking policy, where all nodes have the same duty cycle. Since all nodes blink between the active and inactive state in tandem, all disk state is accessible during the active periods. While the synchronous policy is not necessarily optimal for all applications and maximizes peak power demand, it does eliminate the complexities of dealing with application-specific communication patterns when determining the blink schedule. However, dealing with application-specific complexities may improve performance: an asynchronous blinking policy may reduce costs by reducing peak power or an asymmetric blinking policy may improve performance by prioritizing nodes that store heavily-accessed data.

In general, distributed applications that store per-node state require application modifications to gracefully handle blinking, and ensure the state is available at the proper
Figure 5.2. Hardware architecture of the Blink prototype.

times. However, since intermittent power scenarios may yield little or no power during certain periods, it is not appropriate in all application scenarios. For instance, intermittent power is not appropriate for applications that make performance guarantees. The approach is applicable in many scenarios, such as for best-effort batch jobs, including Hadoop/MapReduce jobs, or for performance optimizations that augment an always-on infrastructure. In this chapter, we focus on the latter example by developing a version of memcached, called BlinkCache, that gracefully handles intermittent power constraints via blinking. While memcached’s design explicitly avoids the type of inter-node dependencies present in Hadoop, and other tightly-coupled distributed applications, it is able to benefit from an asymmetric load-proportional policy, as we describe in Section 5.4. Additionally, nodes dedicated to memcached deployments do not require disks. Mechanical disks pose an additional constraint for blinking due to both performance and reliability concerns with frequently spinning disks up and down [78]. The effect of power cycling on the lifetime of CPUs, memory, and motherboards is an open question; we know of no prior work that addresses the issue.

5.3 Blink Prototype

Blink is a combined hardware/software platform for developing and evaluating blinking applications. This section describes our prototype’s hardware and software architecture in detail.
5.3.1 Blink Hardware Platform

Blink’s current hardware platform consists of two primary components: (i) a low-power server cluster that executes Blink-aware applications and (ii) a variable energy source constructed using an array of micro wind turbines and solar panels. We use renewable energy to expose the cluster to intermittent power constraints.

5.3.1.1 Energy Sources

We deployed an array of two wind turbines and two solar panels to power Blink. Each wind turbine is a SunForce Air-X micro-turbine designed for home rooftop deployment, and rated to produce up to 400 watts in steady 28 mph winds. However, in our measurements, each turbine generates approximately 40 watts of power on windy days. Our solar energy source uses Kyocera polycrystalline solar panels that are rated to produce a maximum of 65 watts at 17.4 volts under full sunlight. Although polycrystalline panels are known for their efficiency, our measurements show that each panel only generates around 30 watts of power in full sunlight and much less in cloudy conditions.

We assume blinking systems use batteries for short-term energy storage and power buffering. Modern data centers and racks already include UPS arrays to condition power and tolerate short-term grid disruptions. We connect both renewable energy sources in our deployment to a battery array that includes two rechargeable deep-cycle ResourcePower Marine batteries with an aggregate capacity of 1320 watt-hours at 12V, which is capable of powering our entire cluster continuously for over 14 hours. However, in this chapter we focus on energy-neutral operation over short time intervals, and thus use the battery array only as a small 5-minute buffer. We connect the energy sources to the battery pack using a TriStar T-60 charge controller that provides over-charging circuitry. We deployed our renewable energy sources on the roof of a campus building in September 2009 and used a HOBO U30 data logger to gather detailed traces of current and voltage over a period of several months under a variety of different weather conditions.

While our energy harvesting deployment is capable of directly powering Blink’s server cluster, to enable controlled and repeatable experiments we leverage two Extech programmable power supplies. We use the programmable power supplies, instead of the harvesting deployment, to conduct repeatable experiments by replaying harvesting traces, or emulating other intermittent power constraints, to charge our battery array.  

Since the battery’s voltage level indicates its current energy capacity, we require sensors to measure and report it. We use a data acquisition device (DAQ) from National Instruments to facilitate voltage measurement. As shown in Figure 5.2, the prototype includes two high-precision 5MOhm resistors between the battery terminals and employs the DAQ to measure voltage across each resistor. We then use the value to compute the instantaneous battery voltage, and hence, capacity. Figure 5.3 shows the empirically-derived capacity of our prototype’s battery as a function of its voltage level. In addition to battery voltage, we use DC current transducers to measure the current flowing from the energy source into the

---

3 We are able to set the initial battery level for each experiment using a separate charge controller in load-control mode.
Figure 5.3. Empirically-measured battery capacity as a function of voltage for our deep-cycle battery. We consider the battery empty below 12V, since using it beyond this level will reduce its lifetime.

battery, and the current flowing from the battery to the cluster. The configuration allows Blink to accurately measure these values every second.

5.3.1.2 Low-power Server Cluster

Our Blink prototype uses a cluster of low-power server nodes. Since our energy harvesting deployment is only capable of producing 100-140 watts of power, a cluster of traditional high-power servers, such as Xeon servers that consume roughly 500W each, is not feasible. As a result, we construct our prototype from low-power nodes that use AMD Geode processor motherboards. Each motherboard, which we scavenge from OLPC-XO laptops, consists of a 433 MHz AMD Geode LX CPU, 256 MB RAM, a 1GB solid-state flash disk, and a Linksys USB Ethernet NIC. Each node runs the Fedora Linux distribution with kernel version 2.6.25. We connect our 10 node cluster together using 2 energy-efficient 8-port Rosewill 100 Mbps switches. Each low-power node consumes a maximum of 8.6W, and together with the switch, the 10 node cluster has a total energy footprint of under 100 watts, which closely matches the energy generated from our renewable energy sources.

Similar clusters of low-power nodes, e.g., using Intel Atom processors, are currently being considered by data centers for energy-efficient processing of I/O-intensive workloads [39]. Our low-power Blink design should also scale to traditional Xeon-class servers for appropriately sized energy sources, although, as we discuss in Section 5.5, an application’s performance may differ for higher-power nodes. An advantage of using XO motherboards is that they are specifically optimized for rapid S3 transitions that are useful for blinking. Further, the motherboards use only 0.1W in S3 and 8.6W in S0 at full processor and network utilization. The wide power range in these two states combined with the relatively low power usage in S3 makes these nodes an ideal platform for demonstrating the efficacy of Blink’s energy optimizations.
### Blinking Interface

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>setDutyCycle(int nodeId, int onPercentage)</td>
</tr>
<tr>
<td>setBlinkInterval(int nodeId, int interval)</td>
</tr>
<tr>
<td>syncActiveTime(int node, long currentTime)</td>
</tr>
<tr>
<td>forceSleep(int nodeId, int duration)</td>
</tr>
</tbody>
</table>

**Table 5.2.** Blink APIs for setting per-node blinking schedules.

### Measurement Interface

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBatteryCapacity()</td>
</tr>
<tr>
<td>getBatteryEnergy()</td>
</tr>
<tr>
<td>getChargeRate(int lastInterval)</td>
</tr>
<tr>
<td>getDischargeRate(int lastInterval)</td>
</tr>
<tr>
<td>getServerLoadStats(int nodeId)</td>
</tr>
</tbody>
</table>

**Table 5.3.** Blink’s measurement APIs that applications use to inform their blinking decisions.

#### 5.3.2 Blink Software Architecture

Blink’s software architecture consists of an application-independent control plane that combines a power management service with per-node access to energy and node-level statistics. Blink-aware applications interact with the control plane using Blink APIs to regulate their power consumption. The power management service consists of a power manager daemon that runs on a gateway node and a power client daemon that runs on each cluster node. The architecture separates mechanism from policy by exposing a single simple interface for applications to control blinking for each cluster node.

The power manager daemon has access to the hardware sensors, described above, that monitor the battery voltage and current flow. Each Blink power client also monitors host-level metrics on each cluster node and reports them to the power manager. These metrics include CPU utilization, network bandwidth, and the length of the current active period. The power client exposes an internal RPC interface to the power manager that allows it to set a node’s blinking pattern. To set the blinking pattern, the power client uses the timer of the node’s real-time clock (RTC) to automatically sleep and wake up, i.e., transition back to S0, at specific intervals. Thus, the power client is able to set repetitive active and inactive durations. For example, the power manager may set a node to repeatedly be active for 50 seconds and inactive for 10 seconds. In this case, the blink interval is 60 seconds with the node being active 83% of the time and inactive 17% of the time. We assume that nodes synchronize clocks using a protocol such as NTP to enable policies that coordinate blink schedules across cluster nodes.

The impact of clock synchronization is negligible for our blink intervals at the granularity of seconds, but may become an issue for blink intervals at the granularity of milliseconds or less. Note that clock synchronization is not an issue for applications, such
as memcached, that do not perform inter-node communication. Transitioning between S0 and S3 incurs a latency that limits the length of the blink interval. Shorter blink intervals are preferable since they allow each node to more closely match the available power, more rapidly respond to changes in supply, and reduces the battery capacity necessary for short term buffering. The XO motherboard yields S3 sleep latencies that range from roughly 200 milliseconds to 2 seconds depending on the set of active devices and drivers (see Table 5.1). For instance, since our USB NIC driver does not implement the ACPI reset_resume function, we must unload and load its driver when transitioning to and from S3. As a result, the latency for our experiments is near 2 seconds. Unfortunately, inefficient and incorrect device drivers are commonplace, and represent one of the current drawbacks to blinking in practice.

The Blink control plane exposes an RPC interface to integrate with external applications as shown in Tables 5.2 and 5.3. Applications use these APIs to monitor input/output current flow, battery voltage, host-level metrics and control per-node blinking patterns. Since Blink is application-independent, the prototype does not report application-level metrics. In such cases, an application must monitor itself. For instance, for some policies in our blinking version of memcached, a proxy monitors per-key hit rates, as described in Section 5.4.3.1.

5.4 Blinking Memcached

Memcached is a distributed in-memory cache for storing key-value pairs that significantly reduces both the latency to access data objects and the load on persistent disk-backed storage. Memcached has become a core component in Internet services that store vast amounts of user-generated content, with services maintaining dedicated clusters with 100s to 1000s of nodes [63]. Since end users interact with these services in real-time through web portals, low-latency access to data is critical. High page load latencies frustrate users and may cause them to stop generating new content [62], which is undesirable since these services’ primary source of revenue derives from their content, e.g., by selling targeted ads.

5.4.1 Memcached Overview

Memcached’s design uses a simple and scalable client-server architecture, where clients request a key value directly from a single candidate memcached server with the potential to store it. Clients use a built-in mapping function to determine the IP address of this candidate server. Initial versions of memcached determined the server using the function Hash(Key)%NumServers, while the latest versions use the same consistent hashing approach popularized in DHTs, such as Chord [71]. In either case, the key values randomly map to nodes without regard to their temporal locality, i.e., popularity. Since all clients use the same mapping function, they need not communicate with other clients or servers to compute which server to check for a given key. Likewise, Memcached servers respond to client requests (gets and sets) without communicating with other clients or servers. This lack of inter-node communication enables Memcached to scale to large clusters.

Importantly, clients maintain the state of the cache, including its consistency with persistent storage. As a result, applications are explicitly written to use memcached by (i)
Figure 5.4. The popularity rank, by number of fans, for all 20 million public group pages on Facebook follows a Zipf-like distribution with $\alpha = 0.6$.

checking whether an object is resident in the cache before issuing any subsequent queries, (ii) inserting a newly referenced object into the cache if it is not already resident, and (iii) updating a cached object to reflect a corresponding update in persistent storage. Each memcached server uses the Least Recently Used (LRU) replacement policy to evict objects. One common example of a cached object is an HTML fragment generated from the results of multiple queries to a relational database and other services. Since a single HTTP request for many Internet services can result in over 100 internal, and potentially sequential, requests to other services [63, 43], the cache significantly decreases the latency to generate the HTML.

5.4.2 Access Patterns and Performance Metrics

The popularity of web sites has long been known to follow a Zipf-like distribution [40, 76], where the fraction of all requests for the $i$-th most popular document is proportional to $1/i^\alpha$ for some constant $\alpha$. Previous studies [40, 76] have shown that $\alpha$ is typically less than one for web site popularity. The key characteristic of a Zipf-like distribution is its heavy tail, where a significant fraction of requests are for relatively unpopular objects. We expect the popularity of user-generated content for an Internet service to be similar to the broader web, since, while some content may be highly popular, such as a celebrity’s Facebook page, most users are primarily interested in either their own content or the content of close friends and associates.

As a test of our expectation, we rank all 20 million user-generated fan pages on Facebook by their number of fans. We use the size of each page’s fan base as a rough approximation of the popularity of its underlying data objects. Figure 5.4 confirms that the distribution is Zipf-like with $\alpha$ approximately 0.6. Recent work also states that Facebook must store a significant fraction of their data set in massive memcached cluster, i.e., on the order of 2000 nodes, to achieve high hit rates, e.g., 25% of the entire data set to achieve a 96.5% hit rate [63]. This characteristic is common for Zipf-like distributions with low $\alpha$ values, since many requests for unpopular objects are inside the heavy tail. Thus, the distribution roughly divides objects into two categories: the few highly popular objects and
the many relatively unpopular objects. As cache size increases, it stores a significant fraction of objects that are unpopular compared to the few popular objects, but nearly uniformly popular compared to each other. These mega-caches resemble a separate high-performance storage tier [63] for all data objects, rather than a small cache for only the most popular data objects.

Before discussing different designs alternatives for BlinkCache, we define our performance metrics. The primary cache performance metric is hit ratio, or hit rate, which represents the percentage of object requests that the cache services. A higher hit rate indicates both a lower average latency per request, as well as lower load on the back-end storage system. In addition to hit rate, we argue that fairness should be a secondary performance metric for large memcached clusters that store many objects of equal popularity. A fair cache distributes its benefits—low average request latency—equally across objects. Caches are usually unfair, since their primary purpose is to achieve high hit rates by storing more popular data at the expense of less popular data. However, fairness increases in importance when there are many objects with a similar level of popularity, as in today’s large memcached clusters storing data that follows a Zipf-like popularity distribution. An unfair cache results in a wide disparity in the average access latency for these similarly popular objects, which ultimately translates to end-users receiving vastly different levels of performance. We use the standard deviation of average request latency per object as our measure of fairness. The lower the standard deviation the more fair the policy, since this indicates that objects have average latencies that are closer to the mean.

5.4.3 BlinkCache Design Alternatives

We compare variants of three basic memcached policies for variable power constraints: an activation policy, a synchronous policy, and an asymmetric load-proportional policy. In all cases, any get request to an inactive server always registers as a cache miss, while any set request is deferred until the node becomes active. We defer a discussion of the implementation details using Blink to the next section.

- **Activation Policy.** An activation policy ranks servers $1 \ldots N$ and always keeps the top $M$ servers active, where $M$ is the maximum number of active servers the current power level supports. We discuss multiple activation variants, including a *static* variant that does not change the set of available servers in each client’s built-in mapping function to reflect the current set of active servers, and a *dynamic* variant that does change the set. We also discuss a *key migration* variant that continuously ranks the popularity of objects and migrates them to servers $1 \ldots N$ in rank order.

- **Synchronous Policy.** A synchronous policy keeps all servers active for time $t$ and inactive for time $T - t$ for every interval $T$, where $t$ is the maximum duration the current power level supports and $T$ is short enough to respond to power changes but long enough to mitigate blink overhead. The policy does not change the set of available servers in each client’s built-in mapping function, since all servers are active every interval.
• **Load-Proportional Policy.** A load-proportional policy monitors the aggregate popularity of objects \( P_i \) that each server \( i \) stores and keeps each server active for time \( t_i \) and inactive for time \( T - t_i \) for every interval \( T \). The policy computes each \( t_i \) by distributing the available power in the same proportion as the aggregate popularity \( P_i \) of the servers. The load-proportional policy also migrates similarly popular objects to the same server.

5.4.3.1 Activation Policy

A straightforward approach to scaling memcached as power varies is to activate servers when power is plentiful and deactivate servers when power is scarce. One simple method for choosing which servers to activate is to rank them \( 1...N \) and activate and deactivate them in order. Since, by default, memcached maps key values randomly to servers, our policy for ranking servers and keys is random. In this case, a static policy variant that does not change each client’s built-in mapping function to reflect the active server set arbitrarily favors keys that happen to map to higher ranked servers, regardless of their popularity. As a result, requests for objects that map to the top-ranked server will see a significantly lower average latency than requests for objects that happen to map to the bottom-ranked server. One way to correct the problem is to dynamically change the built-in client mapping function to only reflect the current set of active servers. With constant power, dynamically changing the mapping function will result in a higher hit rate since the most popular objects naturally shift to the current set of active servers.

**Hash-based Key Mapping.** Memcached recently added support for consistent hashing to reduce the disruption from changing a cluster’s size. The original function \((\text{Hash}(\text{Key}) \% \text{NumServers})\) invalidates nearly every key when adding or removing a single server from a cluster of size \( n \). Only the keys that have both the old \text{NumServers} and the new \text{NumServers} as common factors do not change mappings. The approach is clearly not scalable, since each change, regardless of size, flushes nearly the entire cache, which abruptly increases load on the back-end storage system, as well as request latency, until the cache re-populates itself.

Consistent hashing, originally popularized [71] by DHTs, significantly improves the situation, since adding or removing each server only invalidates \( 1/n^{th} \) of the keys for a cluster of size \( n \). The approach scales gracefully, since the percentage of keys the cache invalidates by changing a single server in the active set decreases as cluster size increases. However, consistent hashing is not a panacea when power varies either frequently or significantly. For instance, if the available power doubles, thereby doubling the number of active servers, a consistent hashing approach will still invalidate 50% of its resident objects. With a dynamic approach, frequent variations in power repeatedly incur this steep invalidation penalty.

Thus, while our dynamic variant results in a higher hit rate than a static variant under constant power, the opposite is true, due to invalidation penalties, for power that varies frequently or significantly. One option for eliminating invalidation penalties entirely is to explicitly control the mapping of individual keys to servers, and pro-actively migrate the most popular objects to the most popular servers. Figure 5.5 illustrates a memcached design that interposes an always-active proxy between memcached clients and servers to control
Figure 5.5. To explicitly control the mapping of keys to servers, we interpose always-active request proxies between memcached clients and servers. The proxies are able to monitor per-key hit rates and migrate similarly popular objects to the same nodes.

the mapping. In this design, clients issue requests to the proxy, which maintains a hash table that stores the current mapping of keys to servers, issues requests to the appropriate back-end server, and returns the result to the client.

**Table-based Key Mapping.** Since all requests pass through the proxy, it is able to continuously monitor and sort the popularity of objects in the background and dynamically change the server mappings as popularities change. Note that to migrate an object the proxy need only change its key→server mapping. After the change, the next key request for the object will incur one cache miss on the new server, which results in application-level code re-generating and re-inserting the object at the new location. The proxy may either pro-actively evict the object from the old server or simply allow LRU replacement to evict the object. This strategy eliminates invalidation penalties, since popularity-based migration always places the most popular objects on the highest-ranked servers. The design requires no changes to either memcached clients or servers.

Deactivating lower-ranked servers invalidates objects that are already less popular than objects on the higher-ranked active servers, while activating additional servers grows the size of the cache without invalidating the more popular objects that are already resident. The approach does introduce the overhead of monitoring and sorting keys by popularity, as well as proxying requests through an intermediary server. However, these overhead costs are independent of power variations and amortized over time, rather than imposed abruptly at each change in the power level. The proxy approach scales to multiple proxies by allowing memcached clients to use their original built-in mapping function to map keys to a set of proxies instead of a set of servers, such that each proxy is responsible for a random subset of keys.

In a multi-proxy design, the proxies periodically send popularity statistics to a central server that sorts them in the background and distributes each proxy a new server mapping. The period between re-mappings is a function of both how fast key popularity is changing and the number of keys in the cache, since drastic popularity changes require re-mappings and more keys increase the time to gather, sort, and distribute new mappings. Changes to server mappings need not be highly synchronized across all proxies, though, since mem-
cached provides no guarantees that data is present or consistent with a back-end database. A multi-proxy design allows the proxies to share the bandwidth and processing load of issuing client requests to servers, while maintaining the ability to explicitly control key mappings.

We expect a few-to-many relationship between proxies and memcached servers, although highly-loaded memcached clusters may require a mapping as low as one-to-one. In this case, the argument for using proxies instead of the nodes themselves mirrors Somniloquy’s argument for using low-power, augmented network interfaces to perform simple application functions on behalf of sleeping nodes [35]. Another consideration for a proxy-based migration approach is the load imposed on the highest-ranked memcached servers. If the highest-ranked servers become overloaded, the re-mapping process may need to place a few less popular keys on high-ranked nodes to ensure they are not overloaded. The technique is analogous to Popularity-based Data Concentration (PDC) for disk arrays that must be careful not to deactivate too many disks and then overload the remaining active disks [65].

5.4.3.2 Synchronous Policy

The migration-enabled activation policy, described above, approaches the optimal policy for maximizing the cache’s hit rate, since ranking servers and mapping objects to them according to popularity rank makes the distributed cache operate like a centralized cache that simply stores the most popular objects regardless of the cache’s size. We define optimal as the hit rate for a centralized cache of the same size as the distributed Memcached instance under the same workload. However, the policy is unfair for servers that store similarly popular objects, since these servers should have equal rankings. The activation policy is forced to arbitrarily choose a subset of these equally ranked servers to deactivate. In this case, a synchronous policy is significantly more fair and results in nearly the same hit rate as the optimal activation policy. To see why, consider the simple 4-node memcached cluster in Figure 5.6 with enough available power to currently activate half the cluster. There is enough power to support either (i) our activation policy with migration that keeps two nodes continuously active or (ii) a synchronous policy that keeps four nodes active half the time but synchronously blinks them between the active and inactive state.

For now we assume that all objects are equally popular, and compare the expected hit rate and standard deviation of average latency across objects for both policies, assuming a full cache can store all objects at full power on the 4 nodes. For the activation policy, the hit rate is 50%, since it keeps two servers active and these servers store 50% of the objects. Since all objects are equally popular, migration does not significantly change the results. In this case, the standard deviation is 47.5ms, assuming an estimate of 5ms to access the cache and 100ms to regenerate the object from persistent storage. For a synchronous policy, the hit rate is also 50%, since all 4 nodes are active half the time and these nodes store 100% of the objects. However, the synchronous policy has a standard deviation of 0ms, since all objects have a 50% hit probability, if the access occurs when a node is active, and a 50% miss probability, if the access occurs when a node is inactive. Rather than half the objects having a 5ms average latency and half having a 100ms average latency, as with activation, a synchronous policy ensures an average latency of 52.5ms across all objects.
Figure 5.6. Graphical depiction of a static/dynamic activation blinking policy (a), an activation blinking policy with key migration (b), and a synchronous blinking policy (c).

Note that the synchronous policy is ideal for a normal memcached cluster with a mapping function that randomly maps keys to servers, since the aggregate popularity of objects on each server will always be roughly equal. Further, unlike an activation policy that uses the dynamic mapping function, the synchronous policy does not incur invalidation penalties and is not arbitrarily unfair to keys on lower-ranked servers.

5.4.3.3 Load-Proportional Policy

A synchronous policy has the same hit rate as an activation policy when keys have the same popularity, but is significantly more fair. However, an activation policy with migration is capable of a significantly higher hit rate for highly skewed popularity distributions. A proportional policy combines the advantages of both approaches for Zipf-like distributions, where a few key values are highly popular but there is a heavy, but significant, tail of similarly unpopular key values. As with our activation policy, a proportional policy ranks servers and uses a proxy to monitor object popularity and migrate objects to servers in rank order. However, the policy distributes the available power to servers in the same proportion as the aggregate popularity of their keys.

For example, assume that in our 4 server cluster after key migration the percentage of total hits that go to the first server is 70%, the second server is 12%, the third server is 10%, and the fourth server is 8%. If there is currently 100W of available power then the first server ideally receives 70W, the second server 12W, the third server 10W, and the fourth server 8W. These power levels then translate directly to active durations over each interval $T$. In practice, if the first server’s maximum power is 50W, then it will be active the entire interval, since its maximum power is 70W. The extra 20W is distributed to the remaining servers proportionally. If all servers have a maximum power of 50W, the first server receives 50W, the second server receives 20W, i.e., 40% of the remaining 50W, the third server receives 16.7W, and the fourth server receives 13.3W. These power levels translate into the following active durations for a 60 second blink interval: 60 seconds, 24 seconds, 20 seconds, and 16 seconds, respectively.
Table 5.4. Summary of the best policy for a given performance metric and workload combination.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Workload</th>
<th>Best Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit Rate</td>
<td>Uniform</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Hit Rate</td>
<td>Zipf</td>
<td>Activation (Migration)</td>
</tr>
<tr>
<td>Fairness</td>
<td>Uniform/Zipf</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Fairness + Hit Rate</td>
<td>Zipf</td>
<td>Load-Proportional</td>
</tr>
</tbody>
</table>

The hit rate from a proportional policy is only slightly worse than the hit rate from the optimal activation policy. In this example, we expect the hit rate from an activation policy to be 85% of the maximum hit rate from a fully powered cluster, while we expect the hit rate from a proportional policy to be 80.2%. However, the policy is more fair to the 3 servers—12%, 10%, and 8%—with similar popularities, since each server receives a similar total active duration. The Zipf distribution for a large memcached cluster has similar attributes. A few servers store highly popular objects and will be active nearly 100% of the time, while a large majority of the servers will store equally unpopular objects and blink in proportion to their overall unpopularity.

5.4.4 Summary

Table 5.4 provides a summary of the best policy for each performance metric and workload combination. In essence, an activation policy with key migration will always have the highest hit rate. However, for distributions with equally popular objects, the synchronous policy achieves a similar hit rate and is more fair. A load-proportional policy combines the best attributes of both for Zipf-like distributions, which include a few popular objects but many similarly unpopular objects.

5.5 Implementation and Evaluation

We implement and evaluate the BlinkCache design alternatives from the previous section using our small-scale Blink prototype. The purpose of our evaluation is not to maximize the performance of our particular memcached deployment or improve on the performance of the custom memcached server deployments common in industry. Instead, our goal is to explore the effects of churn on memcached caused by power fluctuations for different BlinkCache designs. Our results will differ across platforms according to the specific blink interval, CPU speed, and network latency and bandwidth of the servers and the network. Since our prototype uses low-power CPUs and motherboards, the request latencies we observe in our prototype are not representative of those found in high performance servers.

Each node in Blink connects to a low-power (2.4W/switch) 100 Mbps switch and runs an instance of Blink’s power client and an unmodified memcached server. We wrote a memcached client workload generator to issue key requests at a configurable, but steady,
rate according to either a Zipf popularity distribution, parameterized by $\alpha$, or a uniform popularity distribution. As in a typical application, the workload generator fetches any data not resident in the cache from a MySQL database and places it in the cache. Since we assume the MySQL server provides always-available persistent storage, it runs off the power grid and not variable power. Note that for our benchmarks we take the conservative approach of simply fetching key values directly from MySQL with a single query, and do not imitate the sequential, multi-query nature of production web applications. While we do not model or evaluate the impact of workloads that include multi-get memcached requests that issue gets for multiple keys in parallel, blinking with key migration should not impact the performance of multi-get requests for keys with similar popularities, e.g., part of the same HTML page, since our proxy will migrate these keys to the same server.

Unless otherwise noted, in our experiments, we use moderate-size objects of 10 kilobytes, Facebook-like Zipf $\alpha$ values of 0.6, and memcached’s consistent hashing mapping function. Each experiment represents a half-hour trace, we configure each memcached server with a 100MB cache to provide an aggregate cache size of 1GB, and we use our programmable power supply to drive each power trace. Since each node has only 256MB of memory, we scale our workloads appropriately for evaluation. We modify magent, a publicly available memcached proxy,\(^4\) to implement the design alternatives in the previous section, including table-based key mapping and popularity-based key migration. Our modifications are not complex: we added or changed only 300 lines of code to implement all of the BlinkCache design variants from Section 5.3. Since all requests pass through the proxy, it is able to monitor key popularity. The proxy controls blinking by interacting with Blink’s power manager, which in our setup runs on the same node, to monitor the available power and battery level and set per-node blinking patterns. We also use the proxy for experiments with memcached’s default hash-based key mappings, rather than modifying the memcached client. Since our always-on proxy is also subject to intermittent power constraints, we run it on a low-power (5W) embedded SheevaPlug with a 1.2 GHz ARM CPU and 512 MB of memory.

We first use our workload generator to benchmark the performance of each blinking policy for both Zipf-like and uniform popularity distributions at multiple power levels with varying levels of oscillation. We then demonstrate the performance for an example web application—tag clouds in GlassFish—using realistic traces from our energy harvesting deployment that have varying power and oscillation levels.

### 5.5.1 Benchmarks

We measure the maximum power of each node, at 100% CPU and network utilization, in S0 to be 8.6W and its minimum power in S3 to be 0.2W. We use these values in the proxy to compute the length of active and inactive periods to cap power consumption at a specific level. We also measure the impact of our node’s near 2 second transition latency for blink intervals $T$ between 10 seconds and 2 minutes. Figure 5.7 shows the results for a duty cycle of 50%. In this case, the blinking interval must be over 40 seconds before average power over the interval falls below 55% of the node’s maximum power, as we

---

\(^4\)http://code.google.com/p/memagent/
Figure 5.7. The near 2 second latency to transition into and out of S3 in our prototype discourages blinking intervals shorter than roughly 40 seconds. With a 50% duty cycle we expect to operate at 50% full power, but with a blink interval of less than 10 seconds we operate near 100% full power.

Figure 5.8. Maximum throughput (a) and latency (b) for a dedicated memcached server, our memcached proxy, and a MySQL server. Our proxy imposes only a modest overhead compared with a dedicated memcached server.

expect. The result shows that on-demand transitions that occur whenever work arrives or departs are not practical in our prototype. Further, even blinking intervals as high as 10 seconds impose significant power inefficiencies. As a result, we use a blinking interval of 60 seconds for our experiments. Our 60 second blink interval is due solely to limitations in the Blink prototype. Note that there is an opportunity to significantly reduce blink intervals through both hardware and software optimizations. Since server clusters do not typically leverage ACPI’s S3 state, there has been little incentive to optimize its transition latency.

Next, we determine a baseline workload intensity for memcached, since, for certain request rates and key sizes, the proxy or the switch becomes a bottleneck. In our experiments, we use a steady request rate (1000 get requests/sec) that is less than the maximum request rate possible once the proxy or switch becomes a bottleneck. Note that our results, which focus on hit rates, are a function of the popularity of objects rather than the distribution of request inter-arrival times. Our goal is to evaluate how blinking affects the
Figure 5.9. Under constant power for a Zipf popularity distribution, the dynamic variant of the activation policy performs better than the static variant as power decreases. However, the activation policy with key migration outperforms the other variants.

relative hit rates between the policies, and not the performance limitations of our particular set of low-power components. Figure 5.8 demonstrates the maximum performance, in terms of total throughput and request latency for different key values sizes, of an unmodified memcached server, our memcached proxy, and a MySQL server. As expected, the memcached server provides an order of magnitude higher throughput and lower request latency than MySQL. Further, our proxy implementation imposes only a modest overhead to both throughput and latency, although the latency impact of proxy-based redirections will be greater on faster CPUs since less relative request time is spent in the OS and network. Our subsequent experiments focus on request hit rates rather than request latencies, since latencies vary significantly across platforms and workloads. Further, the wide disparity in latency between serving a request from memory and serving it from disk would show a larger, and potentially unfair, advantage for a blinking system. Thus, we consider hit rate a better metric than latency for evaluating a blinking memcached instance.

5.5.1.1 Activation Blinking and Thrashing

An activation policy for an unmodified version of memcached must choose whether or not to alter the hash-based mapping function as it activates and deactivates servers. For constant power, a dynamic mapping function that always reflects the currently active set of servers should provide the best hit rate, regardless of the popularity distribution, since applications will be able to insert the most popular keys on one of the active servers. Figure 5.9 demonstrates this point for a workload with a Zipf popularity distribution ($\alpha = 0.6$), and shows the hit rates for both static and dynamic activation variants at multiple constant power levels. While at high power levels the approaches have similar hit rates, as power level decreases, we see that the static variant incurs a higher penalty under constant power. However, Figure 5.10 demonstrates that the opposite is true for highly variable power. The figure reports hit rates for different levels of power oscillation, where the average power for each experiment is 45% of the power necessary to run all nodes concurrently. The $x$-axis indicates oscillation level as a percentage, such that 0% oscillation holds power steady...
Figure 5.10. Under oscillating power for a Zipf popularity distribution, the static variant of the activation policy performs better than the dynamic variant as the oscillation increases. Again, the activation policy with key migration outperforms the other variants.

throughout the experiment and $N\%$ oscillation varies power between $(45 + 0.45N)\%$ and $(45 - 0.45N)\%$ every 5 minutes.

We see that dynamic changes in the active server set of memcached’s hash-based mapping function incur an invalidation penalty. Since the invalidation penalty does not occur when memcached does not change the mapping function, the static variant provides a significantly better hit rate as the oscillations increase. Although not shown here, the difference with the original modulo approach is much greater, since each change flushes nearly the entire cache. The hash-based mapping function forces a choice between performing well under constant power or performing well under variable power. A table-based approach that uses our proxy to explicitly map keys to servers and uses key migration to increase the priority of popular keys performs better in both scenarios. That is, the approach does not incur invalidation penalties under continuously variable power, or result in low hit rates under constant power, as also shown in Figure 5.9 and Figure 5.10. Note that oscillation has no impact on other policies, e.g. those using key migration or the synchronous policy.

5.5.1.2 Synchronous Blinking and Fairness

While the activation policy with key migration results in the highest hit rate overall, it is unfair when many servers store equally popular objects since the policy must choose some subset of equally popular servers to deactivate. Figure 5.11 quantifies the fairness of the dynamic activation policy, the activation policy with key migration, and the synchronous policy, as a function of standard deviation in average per-object latency, at multiple constant power levels for a uniform popularity distribution where all objects are equally popular. Note that for distributions where all objects are equally popular, key migration is not necessary and is equivalent to using the static variant of hash-based mapping.

The synchronous policy is roughly 2X more fair than the activation policy with key migration at all power levels. While the dynamic hash-based mapping is nearly as fair as the synchronous policy, it has a worse hit rate, especially in high-power scenarios, as
Figure 5.11. For a uniform popularity distribution, both the synchronous policy and the dynamic variant of the activation policy are significantly more fair, i.e., lower standard deviation of average per-object latency, than the activation policy with key migration.

Figure 5.12. For a uniform popularity distribution, the synchronous policy and the activation policy with key migration achieve a similar hit rate under different power levels. Both policies achieve a better hit rate than the dynamic variant of the activation policy.

shown in Figure 5.12. Thus, the synchronous policy, which is more fair and provides lower average latency, is a better choice than any variant of the activation policy for uniform popularity distributions. Note that the key popularity distribution across servers in *every* memcached cluster that uses a hash-based mapping function is uniform, since keys map to servers randomly. Thus, the synchronous policy is the best choice for a heavily-loaded memcached cluster that cannot tolerate the throughput penalty of using proxies.

5.5.1.3 Balancing Performance and Fairness

Activation with key migration results in the maximum hit rate for skewed popularity distributions where some objects are significantly more popular than others, while the synchronous policy results in the best overall performance, in terms of both hit rate and fairness, for uniform popularity distributions. The proportional policy combines the advan-
Figure 5.13. The load-proportional policy is more fair to the unpopular objects, i.e. bottom 80\% in popularity, than the activation policy with key migration for Zip popularity distributions, especially in low-power scenarios.

Figure 5.14. The load-proportional policy has a slightly lower hit rate than the activation policy with key migration.

tages of both and works well for Zipf-like distributions with a few popular objects but a long tail of similarly (un)popular objects, since the long heavy tail in isolation is similar to the uniform distribution. Figure 5.14 shows the hit rate for the proportional policy, the activation policy with migration, and the synchronous policy for a Zipf popularity distribution with $\alpha = 0.6$ at different power levels. The synchronous policy performs poorly, especially at low power levels, in this experiment, since it does not treat popular objects different than unpopular objects.

However, the proportional policy attains nearly the same hit rate as the activation policy at high power levels, since it also prioritizes popular objects over unpopular objects. Even at low power levels its hit rate is over 60\% of the activation policy’s hit rate. Further, the proportional policy is significantly more fair to the many unpopular objects in the distribution. Figure 5.13 reports fairness, in terms of the standard deviation in per-object latency, at different power levels for the unpopular keys, i.e., keys ranked in the bottom 80th percentile of the distribution. The activation policy’s unfairness is nearly 4X worse at
low power levels. Thus, the proportional policy strikes a balance between performance and fairness when compared against both the synchronous and activation policies.

Finally, Figure 5.15 shows how the S3 transition overhead affects our results at a moderate power level. The figure shows that the overhead has only a modest effect on the load-proportional policy’s hit rate. The overhead does not affect the relative fairness of the policies. Note that all of our previous experiments use our prototype’s 2 second transition overhead. A shorter transition overhead would improve our results, and even a longer transition would show some, albeit lesser, benefits.

5.5.2 Case Study: Tag Clouds in GlassFish

While our prior experiments compare our blinking policies for different power and oscillation levels, we also conduct an application case study using traces from our energy harvesting deployment. The experiment provides a glimpse of the performance tradeoffs for realistic power signals. GlassFish is an open source Java application server from Sun,
which includes a simple example application that reuses parts of the Java PetStore multi-tier web application, used in prior research, e.g., [42], to create tag clouds for pets. Tag clouds are a set of weighted tags that visually represent the most popular words on a web page. We modify the default web application to generate HTML for per-user tag cloud pages and cache them in memcached. The data to construct each HTML page comes from a series of 20 sequential requests to a MySQL database.

For these experiments, we measure the latency to load user tag cloud pages, which incorporates MySQL and HTML regeneration latencies whenever HTML pages are not resident in the cache. The MySQL latency for our simple table-based data is typically 30ms per database query. While page load latency follows the same trend as hit rate, it provides a better application-level view of the impact of different policies. Figure 5.16(b) shows the average latency to load user web pages across 40,000 users for our three different policies—activation with key migration, proportional, and synchronous—for a combined solar and wind trace, assuming the popularity of each user’s tag cloud page follows a Zipf distribution with $\alpha = 0.6$. We derive the power signal, shown in Figure 5.16(a), by compressing a 3-day energy harvesting trace to 3 hours.

As expected, the activation policy with key migration and the load-proportional policy exhibit comparable page load latencies at most points in the trace. For this trace, the load-proportional policy is within 15% of the activation policy’s hit rate. The activation policy is slightly better at low energy levels, since it tends to strictly ensure that more popular content is always cached. Also as expected, the synchronous policy tends to perform poorly across all power levels. Also as expected, we measure the standard deviation of page load latencies for the load-proportional policy to be within 2% to the synchronous policy for the vast majority, i.e., bottom 80%, of the equally unpopular objects.

5.6 Related Work

The sensor network community has studied strategies for dealing with variable sources of renewable power, since these systems often do not have access to the power grid. However, since sensor networks are geographically distributed, each node must harvest its own energy, resulting in network-wide energy imbalances [44], whereas data center nodes share a common power delivery infrastructure. Further, the primary performance metric for a sensor network is the amount of data the network collects. As a result, much of the energy harvesting work is not directly applicable to data centers. Similarly, mobile computing generally focuses on extending battery life by regulating power consumption [77], rather than modulating performance to match energy production.

The increasing energy consumption of data centers [34] has led companies to invest heavily in renewable energy sources [59, 72]. For example, the goal of Google’s RE<C initiative is to make large-scale renewable power generation cheaper than coal-based production. As a result, researchers have started to study how to incorporate renewables into a data center’s power delivery infrastructure [70]. As one example, Lee et al. [50] use request redirection to control the carbon footprint of data centers by redirecting load to servers powered by renewable energy sources. While not directly related to energy harvesting, Power Routing [64] proposes shuffled power delivery topologies that allow data centers
to control how much power each rack receives. While the topologies are well-suited for delivering variable amounts of power to racks based on aggregate demand, they are also useful for flexible routing of a variable power supply. Prior research on workload-driven approaches to improve data center energy efficiency is orthogonal to our work. Examples include designing platforms that balance CPU and I/O capacity [39, 69], routing requests to locations with the cheapest energy [66], and dynamically activating and deactivating nodes as demand rises and falls [41, 74, 49]. PowerNap’s node-level energy proportional technique has also been viewed as a workload-driven optimization [54]. We show that a similar technique is useful for controlling per-node power consumption in a power-driven system.

Power capping has also been studied previously in data centers to ensure collections of nodes do not exceed a worst-case power budget [67, 45]. However, the work assumes exceeding the power budget is a rare transient event that does not warrant application-specific modifications, and that traditional power management techniques, e.g., DVFS, are capable of enforcing the budget. These assumptions may not hold in many scenarios with intermittent power constraints, as with our renewable energy power source. Gandhi et al. cap CPU power by forcing CPU idle periods [46]. While similar, blinking focuses on capping per-node power where the CPU is only one component of the total power draw. Improving the energy-efficiency of storage is also a related research area. While Memcached does not offer persistent storage, our modifications for blinking adapt similar ideas from prior storage research, such as migrating popular objects to more active nodes [65, 78]. Additionally, power-aware caching algorithms focus on maximizing the idle time between disk accesses to reduce disk power consumption, while our work focus on controlling the power consumption of the cache itself [82].

Blinking introduces regulated churn into data center applications as nodes switch from the active to inactive state. Churn has been well-studied in decentralized, self-organizing distributed hash tables [71]. However, the type of churn experienced by DHTs is different than the churn caused by blinking, which motivates our different approach to the problem. In the former case, nodes arrive and depart unexpectedly based on autonomous user behavior and network conditions, while in the latter case, nodes switch between the active and inactive states in a regular and controllable fashion. Finally, RAMCloud [63] proposes using memory for low-latency persistent storage, and cites as motivation the increasingly large memcached clusters used in production data centers. The size of these clusters motivates our observation that fairness for the large number of equally unpopular objects, in addition to hit rate, is an important performance metric.

5.7 Conclusion

In this chapter, we focus on managing server clusters running on intermittent power. We propose blinking as the primary abstraction for handling intermittent power constraints, and define multiple types of blinking policies. We then design an application-independent platform for developing and evaluating blinking applications, and use it to perform an in-depth study of the effects of blinking on one particular application and power source: memcached operating off renewable energy. We find that while an activation policy with key migration results in the best hit rates, it does not distribute the benefits of the cache equally
among equally popular objects. As in-memory caches continue grow in size, they will store a greater fraction of equally popular objects for Zipf-like object popularity distributions. We then propose and evaluate an asymmetric load-proportional policy to increase fairness without significantly sacrificing the cache’s hit rate.
CHAPTER 6

DISTRIBUTED FILE SYSTEM FOR INTERMITTENT POWER

As discussed in the previous chapter, designing systems to operate under intermittent power is challenging, since applications often access persistent distributed state, where power fluctuations can impact data availability and I/O performance. To support concurrent stateful applications on a blinking cluster, in this chapter, I design a blink-aware distributed file system, which combines blinking with a power-balanced data layout and popularity-based replication/reclamation to optimize I/O throughput and latency as power varies.

6.1 Background and Motivation

The growth of cloud-based services continues to fuel a rapid expansion in the size and number of data centers. The trend only exacerbates the environmental and cost concerns associated with data center power usage, which a recent report estimates at 1.7-2.2% of U.S. consumption [124]. Excessive energy consumption also has serious environmental ramifications, since 83% of U.S. electricity derives from burning “dirty” fossil fuels [125]. Energy costs are also on a long-term upward trend, due to a combination of government regulations to limit carbon emissions, a continuing rise in global energy demand, and dwindling supplies. Even with today’s “cheap” power, a data center’s energy-related costs represent a significant fraction (~31% [112]) of its total cost of ownership. Prior research on green data centers often assumes that grid energy is always available in unlimited quantities [110, 111], and focuses largely on optimizing applications to use less energy without impacting performance. By comparison, there has been little research on optimizing applications for intermittent power that fluctuates over time. Operating off intermittent power introduces new opportunities for optimizing a data center to be both cheaper and greener.

The ability to use intermittent power introduces other opportunities, beyond increasing use of renewable energy, for optimizing a data center to be cheaper, greener, and more reliable. We argue that designing systems to exploit these optimizations will move us closer to the vision of a net-zero data center.

• Market-based Electricity Pricing. Electricity prices vary continuously based on supply and demand. Many utilities now offer customers access to market-based rates that vary every five minutes to an hour [107]. As a result, the power data centers are able to purchase for a fixed price varies considerably and frequently over time. For instance, in the New England hourly wholesale market in 2011, maintaining a fixed $55/hour budget, rather than a fixed per-hour power consumption, purchases 16% more power for the same price (Figure 6.1). The example demonstrates that
Figure 6.1. Electricity prices vary every five minutes to an hour in wholesale markets, resulting in the power available for a fixed monetary budget varying considerably over time.

data centers that execute delay-tolerant workloads, such as data-intensive batch jobs, have an opportunity to reduce their electric bill by varying their power usage based on price.

- **Unexpected Blackouts or Brownouts.** Data centers often use UPSs for backup power during unexpected blackouts. An extended blackout may force a data center to limit power consumption at a low level to extend UPS lifetime. While low power levels impact performance, it may be critical for certain applications to maintain some, even low, level of availability, e.g., disaster response applications. As we discuss, maintaining availability at low power levels is challenging if applications access distributed state. Further, in many developing countries, the electric grid is highly unstable with voltage rising and falling unexpectedly based on changing demands. These “brownouts” may also affect the power available to data centers over time.

- **100% Power Infrastructure Utilization.** Another compelling use of intermittent power is continuously operating a data center’s power delivery infrastructure at 100%. Since data center capital costs are enormous, maximizing the power delivery infrastructure’s utilization by operating as many servers as possible is important. However, data centers typically provision power for peak demands, resulting in low utilization [114, 123]. In this case, intermittent power is useful to continuously run a background workload on a set of servers—designed explicitly for intermittent power—that always consume the excess power PDUs are capable of delivering. Since the utilization (and power usage) of a data center’s foreground workload may vary rapidly, the background servers must be capable of quickly varying power usage to not exceed the power delivery infrastructure’s limits.

In this chapter, we present the design of a distributed file system (DFS) for intermittent power. Since DFSs now serve as the foundation for a wide range of data-intensive applications run in today’s data centers, taking advantage of any of the opportunities above necessitates a DFS optimized for intermittent power. As we discuss, designing such a DFS
poses a significant research challenge, since periods of scarce power may render data inaccessible, while periods of plentiful power may require costly data layout adjustments to scale up I/O throughput.

Our work is the first to address the problem by designing a blink-aware stateful DFS optimized for intermittent power. Our system, called BlinkFS, represents a dramatic departure from all prior techniques used in energy-efficient storage systems, which generally rely on powering down large sets of servers for long periods of time to reduce overall energy consumption. Our goal is to design a DFS that performs well across a wide range of intermittent power scenarios—ranging from large and rapid power variations (+/- 90%) to sustained low power periods (∼20%). In all cases, BlinkFS’s goal is to utilize intermittent power as efficiently as possible, rather than guarantee a specific amount of work finishes within some time period. Since intermittent power may not be available to satisfy workloads with strict performance requirements or deadlines, it is not appropriate in these cases. Since we focus on using intermittent power, optimizations that use grid energy in combination with renewables to provide SLA guarantees are beyond the scope of this thesis. As we describe in Section 6.2, the dynamics of intermittent power, where changes in available power may be significant, frequent, and unpredictable, warrant our new approach. Below, we highlight the advantages of our DFS designed for intermittent power, called BlinkFS, over co-opting prior energy-efficient storage techniques, e.g., [38, 48, 126, 65, 129, 78].

• **Low Amortized Overhead.** Blinking every node at regular intervals prevents costly and abrupt data migrations—common in many systems—whenever power decreases—to concentrate data on a small set of active nodes—or increases—to spread data out and increase I/O throughput. Instead, blinking ensures that each node is active, and its data is accessible, for some period of time each blink interval, at the expense of a modest overhead to transition each node between a high-power active and low-power inactive state.

• **Bounded Replica Inconsistency.** Deactivating nodes for long periods requires write off-loading to temporarily cache writes destined for inactive or overloaded nodes [117]. The technique requires excessive writes whenever nodes activate or deactivate to either apply or migrate off-loaded writes, respectively, while impacting reliability if off-loaded writes are lost due to a node failure. In contrast, BlinkFS ensures all replicas are consistent within one blink interval of any write, regardless of the power level.

• **No Capacity Limitations.** Since migrating to a new data layout is expensive, a goal of BlinkFS is to decouple I/O performance at any power level from the data layout: the same layout should perform well at all power levels. To ensure such a data layout, Rabbit [38] severely limits the capacity of nodes storing secondary, tertiary, etc. replicas. Blinking enables a power-independent layout without such limitations.

• **Minimally Disruptive.** DFSs support higher-level applications designed assuming fully active nodes with stable data layouts. Frequently changing the set of active
nodes or the data layout disrupts scheduling and placement algorithms for applications, such as MapReduce, that co-locate computation with DFS storage. BlinkFS is less disruptive, since it keeps every node active for the same duration every blink interval and does not change the data layout as power varies.

- **Always-accessible Data.** Prior systems render data completely inaccessible if there is not enough power to store all data on the set of active nodes. In contrast, BlinkFS ensures all data is accessible, with latency bounded by the blink interval, even at low power levels where the set of active nodes is unable to store the entire data set.

Since each node’s data is inaccessible for some period each blink interval, BlinkFS’s goal is to gain the advantages above without significantly degrading access latency. In achieving this goal, this chapter makes the following contributions.

**Blinking-aware File System Design.** We detail BlinkFS’s design and its advantages over co-opting existing energy-proportional DFSs for intermittent power. The design leverages a few always-active proxies to absorb file system operations, e.g., reads and writes, while masking BlinkFS’s complexity from applications.

**Latency Reduction Techniques.** We discuss techniques for mitigating blinking’s latency penalty. Our approach combines staggered node active intervals with a power-balanced data layout to ensure replicas stored on different nodes are active for the maximum duration each blink interval. BlinkFS also uses popularity-based data replication and reclamation to further decrease latency for frequently-accessed data blocks.

**Implementation and Evaluation.** We implement BlinkFS on a small-scale prototype using 10 Mac minis connected to a programmable power supply that drives variable power traces. We then benchmark BlinkFS’s performance and overheads at different (fixed and oscillating) power levels. We also compare BlinkFS with prior energy-efficient DFSs in two intermittent power scenarios—maintaining a fixed-budget despite variable prices and using intermittent wind/solar energy—using three different applications: a MapReduce-style batch system, the MemcacheDB key-value store, and file system traces from a search engine. As an example of our results, BlinkFS improves MapReduce job completion time by 42% at 50% power compared to an existing energy-proportional DFS. At 20% power, BlinkFS still finishes jobs, while existing approaches stall completely due to inaccessible data.

## 6.2 DFSs and Intermittent Power

Reducing data center power consumption is an active research area. Much prior work focuses on energy-proportional systems, where power usage scales linearly with workload demands [3]. The goal of energy-proportional systems is to not impact performance: if demands increase, these systems increase power consumption to maintain performance. Energy-proportional distributed applications typically vary power consumption by activating and deactivating nodes as workload demands change. An obvious approach for addressing intermittent power is to co-opt existing energy-proportional approaches, but vary the number of active nodes in response to changes in available power rather than workload demands. Unfortunately, as we discuss below, the approach does not work well for DFSs.
using intermittent power, since power variations may be significant, frequent, and unpredictable, e.g., from changing prices, explicit demand response signal sent by the electric grid, or wind, solar, geothermal, etc. power. While energy-proportional systems optimize energy consumption to satisfy workload demands, designing for intermittent power requires systems to optimize performance as power varies. Below, we summarize how intermittent power affects energy-proportional DFSs, and then discuss two specific approaches.

6.2.1 Energy-Proportional DFSs

DFSs, such as the Google File System (GFS) [122] or the Hadoop Distributed File System (HDFS) [68], distribute file system data across multiple nodes. Designing energy-proportional DFSs is challenging, in part, since naïvely deactivating nodes to reduce energy usage has the potential to render data inaccessible [126]. One simple way to prevent data on inactive nodes from becoming inaccessible is by storing replicas on active nodes. Replication is already used to increase read throughput and reliability in DFSs, and is effective if the fraction of inactive nodes is small. For example, with HDFS’s random placement policy for replicas, the probability that any block is inaccessible is \(\frac{m!}{n!} \left(\frac{n-k}{m-k}\right)\) for \(n\) nodes, \(m\) inactive nodes, and \(k\) replicas per block. Figure 6.2 plots the fraction of inaccessible data as a function of the fraction of inactive nodes, and shows that nearly all data is accessible for small numbers of inactive nodes. However, the fraction of inaccessible data rises dramatically once half the nodes are inactive, even for aggressive replication factors, such as \(k=7\). Further, even a few inactive nodes, where the expected percentage of inaccessible data is small, may pose problems, e.g., by stalling batch jobs dependent on a small portion of the inaccessible data.

Thus, a popular approach for designing energy-efficient storage systems is to use concentrated data layouts, which deactivate nodes without causing inaccessible data. The layouts often store primary replicas on one subset of nodes, secondary replicas on another mutually-exclusive subset, tertiary replicas on another subset, etc., to safely deactivate non-primary nodes [38, 126]. Other systems concentrate data to optimize for skewed access patterns, by storing only popular data on a small subset of active nodes [116, 121, 48, 65, 78]. Unfortunately, concentrated layouts cause problems if available power varies independently of workload demands. Below, we highlight three problems with approaches that use concentrated data layout to deactivate nodes for long periods.

Inaccessible Data. If there is not enough power available to activate the nodes necessary to store all data, then some data will become inaccessible at low power levels. As we mention in Section 6.1, sustained low power periods are common in many intermittent power scenarios, such as during extended blackout or brownout periods, when using on-site solar generation on a cloudy day, or maintaining a fixed power budget as energy prices rise. Thus, gracefully degrading throughput and latency down to extremely low power levels is important. With concentrated data layouts, as data size increases, the number of nodes, and hence minimum power level, required to store all data and keep it accessible increases.

Write Off-loading Overhead. Energy-proportional systems leverage write off-loading to temporarily cache writes on currently active nodes, since clients cannot apply writes to inactive nodes, e.g., [117, 38, 129]. Write off-loading is also useful for deferring writes
to overloaded nodes, which are common when only a small number of active nodes store all data. While a small number of active primary nodes decreases the minimum power level necessary to keep data accessible, it overloads primaries by requiring them to process all writes. The approach also imposes abrupt overheads when activating or deactivating nodes, either to apply off-loaded writes to newly active nodes or overloaded primary nodes, respectively. Further, intermittent sources, e.g., wind power, that exhibit abrupt power variations require near immediate node deactivations, precluding the completion of time-consuming operations. While using a large battery array as a buffer mitigates the impact of sudden variations, it is prohibitively expensive [119]. Further, deferring writes to replicas on inactive nodes degrades reliability in the event of node failure. Failure’s consequences are worse during low power periods, by increasing the number of off-loaded writes on active nodes, and the time replicas on inactive nodes remain in an inconsistent state.

**Disrupts Higher-level Application.** A common paradigm for DFSs in data centers is to distribute file system data across compute nodes that host a variety of distributed applications. These applications, e.g., MapReduce, have their own, often highly optimized, algorithms to schedule and place computation to minimize data transfer overheads. Thus, activating and deactivating nodes or changing data layouts as power varies often requires significant modifications to higher-level applications.

Below, we outline two approaches to energy-proportional DFSs that use concentrated data layouts and vary power by activating and deactivating nodes. We highlight the additional problems these DFSs encounter if power variations are significant and frequent.

### 6.2.2 Migration-based Approach

We classify any approach that varies power consumption by migrating data to concentrate it on a set of active nodes, and then deactivating the remaining nodes, as a migration-based approach. With this approach, power variations trigger changes to number of nodes storing either the most popular data or primary, secondary, tertiary, etc. replicas. In either case, data layout changes require migrations to spread data out to provide higher I/O throughput (as nodes become active) or to concentrate data and keep it accessible (as nodes become inactive). Thus, mitigating migration overheads is a focus of prior work on energy-efficient storage [48, 65, 78].
To highlight the problems with this approach, consider the simple example in Figure 6.3(a), where there is enough power to operate four nodes storing primary replicas and the data fills two nodes’ storage capacity. A sudden and unexpected drop in power by 2X, leaving only two active nodes, may not afford enough time for the necessary migrations, leaving some data inaccessible. Even with sufficient time for migration, an additional 2X power drop, leaving only one active node, forces at least 50% of the data to become inaccessible. Note that we focus on regulating power consumption within a single data center. Another way to handle power variations is to migrate applications and their data to remote data centers with ample or cheap power [118]. The technique is infeasible for large storage systems. Even assuming dedicated high-bandwidth network links, we view frequent transfers of large, e.g., multi-petabyte, storage volumes as impractical.

### 6.2.3 Equal-Work Approach

Amur et al. propose an energy-proportional DFS, called Rabbit, that eliminates migration-related thrashing using an *equal-work* data layout [38]. The layout uses progressively larger replica sets, e.g., more nodes store \((n + 1)\)-ary replicas than \(n\)-ary replicas. Specifically, the layout orders nodes \(1 \ldots i\) and stores \(b_i = \frac{B}{i}\) blocks on the \(i\)th node, where \(i > p\) and \(p\) nodes store primary replicas (assuming a data size of \(B\)). The layout ensures that any \(1 \ldots k\) active nodes (for \(k < i\) total nodes) are capable of servicing \(\frac{B}{k}\) blocks, since \(\frac{B}{i} < \frac{B}{k}\). Since the approach is able to spread load equally across any subset of nodes in the ideal case of reading all data, it ensures energy-proportionality with no migrations.

Amur et al. provide details of the approach in prior work [38], including its performance for workloads that diverge from the ideal. Rabbit’s primary constraint is its storage capacity limitations as \(i \rightarrow \infty\), since \(\frac{B}{i}\) defines the capacity for node \(i\). Thus, for \(N\) homogeneous nodes capable of each storing \(M\) blocks, the nodes’ aggregate storage capacity is \(MN\), while Rabbit’s storage capacity is \(pM + \sum_{i=p+1}^{N} \frac{pM}{i} = O(\log N)\). For example, for \(N=500\) nodes and \(M=2^{14}=16384\) 64MB blocks, the aggregate storage capacity across all nodes
is $MN=500$ terabytes, while Rabbit’s capacity is less than 15 terabytes, or 3% of total capacity, when $p=2$.

The relationships above show that the fraction of unused capacity increases linearly with $N$. Thus, the total storage capacity is capable of accommodating significantly more replicas than Rabbit uses as $N$ increases. To reduce capacity limitations, Rabbit is able to individually apply the layout to multiple distinct data sets, by using a different $1 \ldots i$ node ordering for each data set. However, multiplexing the approach between data sets trades off desirable energy-efficient properties, e.g., few nodes storing primary replicas and ideal energy-proportionality. Thus, Rabbit’s design presents issues for large clusters of nodes with similar storage capacities.

### 6.3 Applying Blinking to DFSs

The systems in the previous section use activation policies that vary power consumption only by varying the number of active nodes. As discussed in Chapter 5, the blinking abstraction supports many other types of blinking policies. As we discuss in Section 6.4, BlinkFS uses an asynchronous staggered blinking policy.

To see the advantages of blinking for DFSs, recall the previous section’s example (Figure 6.3(b)), where there is initially enough power to operate four nodes that each provide storage for a fraction of the data. If the available power decreases by 2X, with blinking we have the option of keeping all four nodes active for time $t_{active} = \frac{t}{2}$ every blink interval $t$. In this case, instead of migrating data and concentrating it on two active nodes, we are able to keep the same data layout as before without changing our aggregate I/O throughput over each blink interval, assuming each node has the same I/O throughput when active. Thus, at any fixed power level, blinking is able to provide the same I/O throughput, assuming negligible transition overheads, as an activation approach. However, blinking has a distinct advantage over a migration-based approach if the available power changes, since it is possible to alter node active intervals nearly instantly to match the available power without the overhead of migration. Additionally, in contrast to Rabbit, the blinking approach does not require severe capacity limitations on nodes to maintain throughput. Finally, the approach is beneficial at low power levels if not enough nodes are active to store all data, since data is accessible for some period each blink interval.

### 6.4 BlinkFS Design

Figure 6.4 depicts BlinkFS’s architecture, which resembles other recent DFSs, including GFS [122], HDFS [68], Rabbit [38], etc., that use a master meta-data server to coordinate access to each node’s data via a block server. The master also maintains the file system namespace, tree-based directory structure, file name $\rightarrow$ blocks mapping, and block $\rightarrow$ node mapping, as well as enforces the access control and block placement and replication policy. As in prior systems, files consist of multiple fixed-size blocks replicated on zero or more nodes. To mitigate the impact of node failure, the master may recover from meta-data information stored at one or more proxies, described below, or maintain an up-to-date copy of its meta-data on backup nodes.
BlinkFS also includes a power manager that monitors available power, as well as any energy stored in batteries, using hardware sensors. The power manager implements a blinking policy that continuously alters per-node blinking patterns to match power consumption with available power. Specifically, the power manager communicates with a power client on each node to set the blink interval duration \( t \), as well as its start time and active interval \( t_{\text{active}} \). The power client also acts as an interface for accessing other resource utilization statistics, including CPU utilization, I/O accesses, etc. The power manager informs the master and proxies, described below, of the current blinking policy, i.e., when and how long each node is active every blink interval, and per-node resource utilization statistics.

To access the file system, higher-level applications interact with BlinkFS clients through well-known file system APIs. Our prototype uses the POSIX API’s file system calls.

We do not assume that BlinkFS clients are always active, since clients may run on blinking nodes themselves, e.g., in clusters that co-locate computation and DFS storage. Thus, to enable clients to read or write blocks on inactive nodes, BlinkFS utilizes one or more always-active proxies to intercept read and write requests if a client and block server are not concurrently active, and issue them to the appropriate node when it next becomes active. Each proxy maintains a copy (loaded on startup by querying the master) of the metadata information necessary to access a specific group of files (each file is handled by a single proxy), and ensures replica consistency every blink interval. The proxy propagates any file system operations that change meta-data information to the master before committing the changes. The power manager also maintains an up-to-date view of each node’s power state, since each power client sends it a status message when transitioning to or from the inactive state. The messages also serve as heartbeats: if the power manager does not receive any status messages from a power client within some interval, e.g., 5 minutes, it checks if its block server has failed. A failure prompts the master to initiate recovery actions.

Similar to a set of always-active nodes storing primary replicas, proxies consume power that increases the minimum threshold required to operate the cluster. Importantly, however, proxies only serve as intermediaries, and do not store data. As a result, the data set size does not dictate the number of proxies. However, proxies do limit I/O throughput by redirecting
communication between many clients and block servers through a few points. However, as we discuss below, mostly-active clients may often bypass the proxies when accessing data. Further, proxies are most useful at low power levels, where available power, rather than proxy performance, limits I/O throughput. Below we discuss the details of how BlinkFS’s components facilitate reading and writing files, and then present techniques for mitigating BlinkFS’s high latency penalty.

### 6.4.1 Reading and Writing Files

Proxies mask the complexity of interacting with blinking nodes from applications. The master and each client use a well-known hash function to map a file’s absolute path to a specific proxy. To read or write a file, clients either issue requests to the proxy directly, or use an optimization, discussed below, that bypasses a file’s proxy if the client is active at the same time as the file’s block servers.

**Handling Reads.** The meta-data necessary to read a file includes its block IDs and their version numbers, as well as the (IP) address and blinking information of the block servers storing replicas of the file’s blocks. The proxy holds read requests until a node storing the block becomes active, issues the request to the block server, receives the data, and then proxies it to the client. If multiple block servers storing the block’s replicas are active, the proxy issues the request to the node with the longest remaining active interval, assuming the remaining active time exceeds a minimum threshold necessary to read and transmit the block. Using a proxy to transfer data is necessary when executing both clients and block servers on blinking nodes, since the client may not be active at the same time as the block server storing the requested data.

To optimize reads, mostly-active clients may directly request from the proxy the block information—IDs and version numbers—and blinking policy for each block server holding a replica, and then access block servers directly when they become active. The optimization significantly reduces the proxy load for read-intensive workloads. To ensure the proxy applies all previous client writes to a block before any subsequent reads, the proxy includes
a version number for each block, incremented on every update, in its response to the client. If the version number for the block stored at the block server is lower than the requested version number, then the proxy holds pending writes that it has not yet applied. In this case, the read stalls until the proxy applies the writes and the version numbers match. If the block server has an equivalent or higher version number, it sends back the data immediately. In either case, a block server ensures that a client never gets stale data, i.e., a block of version number lower than the requested version number. Like a Unix file system, application-level file locking might be necessary to ensure the atomicity of cross-block reads, e.g., as in the case of concurrent producers and consumers.

**Handling Writes.** The proxy performs a similar sequence for writes. All writes flow through a file’s proxy, which serializes concurrent writes and ensures all block replicas are consistent each blink interval. The proxy may also return to the client before applying the write to every block replica, since subsequent reads either flow through the proxy or match version numbers at the block server, as described above. The proxy maintains an in-memory write-ahead log to track pending off-loaded writes from clients. Since the log is small, the proxy stores in-memory backups on one or more nodes (updated on each write before returning to the client), which it recovers from after a failure. When the client issues the write, the proxy first records the request in its log, increments the version number of the updated blocks, updates the master metadata and its own metadata, and returns to the client; next it then propagates the write to all replicas as the block servers become active; finally, when all replicas successfully apply the write, it removes the request from its log of pending writes.

Since all block servers are active for a period each blink interval, all replicas are consistent within one blink interval from when the write is issued, and the maximum time a write remains pending in the proxy’s log is one blink interval. Of course, the proxy does have a fixed-size log for pending writes. After filling the log, further write requests stall until the proxy propagates at least one of its queued writes to each replica. Based on available power and the CPU and network utilization of block servers, the proxy limits write throughput to ensure all pending writes are applied within a blink interval, e.g., by stalling additional writes.

As with reads, mostly-active clients could also interact directly with block servers, as long as the client and block server are both active at the same time. In this case, the proxy maintains an intermediate version number for each block, not visible to read requests, to handle concurrent writes. An intermediate version number is always greater than or equal to the real version number for any block. An intermediate version number greater than the real version number indicates that one or more writes are pending for the block. Below we describe the complete flow of a direct or bypass write:

1. To write data directly to block servers a client first sends the filename, offset, and data size (in bytes) to the proxy.
2. The proxy increments the intermediate version number of the blocks to be updated, and sends back the meta-data to the client. The meta-data includes block IDs and their intermediate version numbers, as well as the address and blinking information of block servers storing any replicas.
3. The client pushes the data to all replicas as the block servers become active. Each block server keeps the data from the client in an internal cache until it is directed by the proxy to apply the write or delete it from the cache. The client can push data in any arbitrary order.

4. Once the data is successfully pushed to all the block replicas, the client sends a request to the proxy. The request describes the update (block IDs, version numbers, offsets, data sizes) sent to the replicas. Note that the version number in a block update is same as the intermediate version number assigned by the proxy for the block.

5. The proxy updates the metadata, including the version number, of the blocks and the file metadata, updates the master metadata, and finishes the write operation by returning back to the client. The client then returns back to the application.

6. The proxy notifies block servers to apply writes to blocks.

7. If the client could not finish the write operation within a time threshold set by the proxy, based on the blink and I/O rates of the block servers, the proxy aborts the write and directs the block servers to remove any writes from their caches.

A write operation could span several blocks. To ensure consistency and allow concurrent writes the proxy imposes two restrictions. First, the proxy cannot finish a bypass write operation (steps 5 and 6) until all previous overlapping write operations are already finished or aborted. Second, the proxy stalls a via-proxy write until all previous overlapping bypass writes are either completed or aborted. Two write operations are overlapping if they have at least one block in common. Since versioning and metadata updates are serialized by the proxy, all replicas apply concurrent writes in the same serial order, although the data could arrive in any order. Finally, by applying the restrictions above the proxy also ensures the atomicity of cross-block write operations.

Since a write call in an application returns success only after the proxy updates the metadata information, as described above, a subsequent read call from the same application will always see the written data or a more recent version. Likewise, a read request never gets inconsistent data since it cannot see intermediate versions and all stable versions are already consistent.

**Proxy Overhead and Scalability.** As BlinkFS scales, it requires more proxies to increase its maximum workload, especially at moderate power levels. Note that the workload, and not data size, dictates the number of proxies. At high power, since clients can bypass proxies, proxies are not a bottleneck. At low power, the lack of node availability is the constraint, and not the proxies. For moderate power levels, our experiments (Section 6.6) show a proxy-to-block server ratio of 1:10 performs well, and also suggests that for some workloads a higher ratio may be acceptable. Thus, we expect the power overhead of proxies (and the minimum power necessary for operation) to be less than or equal to 10% in today’s clusters.

### 6.4.2 Reducing the Latency Penalty

While migration-based approaches incur high overheads when power levels change, they ensure data is accessible, i.e., stored on an active node, as long as there is power to
activate nodes necessary to store all data. In contrast, naive blinking incurs a high latency penalty, since each node is inactive for some time each blink interval. BlinkFS combines three techniques to reduce latency: an asynchronous staggered blinking policy, a power-balanced data layout, and popularity-aware replication and reclamation.

**Asynchronous Staggered Blinking.** Staggered blinking’s goal is to minimize the overlap in node active intervals by staggering start times equally across each blink interval. Figure 6.5(a) depicts an example of staggered blinking. To perform well at both high and low power levels, the policy assigns equal-sized active intervals to all nodes, while varying the size of this interval to adjust to changes in available power. Thus, at any power level all nodes are active for the same amount of time. In contrast, while activating all nodes in tandem (akin to co-scheduling) may exhibit slightly lower latencies at high power levels (especially for read requests issued during an active interval that span multiple blocks stored on multiple nodes), it performs much worse at moderate-to-low power since it does not take advantage of replication to reduce latency.

Formally, for available power $p_{\text{available}}$, total power $p_{\text{total}}$ necessary to activate all nodes, total power $p_{\text{inactive}}$ required to keep all nodes in the inactive state, blink interval duration $t$, and $N$ nodes, the duration of each node’s active interval is $t_{\text{active}} = \frac{p_{\text{available}} - p_{\text{inactive}}}{p_{\text{total}} - p_{\text{inactive}}} t$, and the blink start time (within each interval) for the $i$th node (where $i=0\ldots N-1$) is $b_{\text{start}} = (t - t_{\text{active}}) \ast \frac{i}{N-1}$. Next we discuss how combining staggered blinking with a data layout that spreads replicas across nodes, maximizes the time at least one block replica is stored on an active node each blink interval. Importantly, the approach maximizes this time at all power levels.

**Power-balanced Data Layout.** A power-balanced data layout spreads replicas for each block across nodes, such that any set of nodes storing the block’s replicas have minimum overlapping active intervals using the staggered blinking policy above. To place replicas in such a layout, we order all $N$ nodes in a circular chain from $0\ldots N-1$ and choose a random node to store the first replica of each block. We then place the second replica on the node opposite the first replica in the circle, the third replica on one of the nodes half-way between the first and second replicas, the fourth replica on the other node between the first and second replicas, etc. To delete replicas, we reverse the process. Figure 6.5(b) depicts an example for three replicas using staggered blinking from 6.5(a).

The layout policy above is optimal, i.e., maximizes the time each block is available on an active node each blink interval, if the number of replicas is a power of two. Maintaining an optimal placement for any number of replicas requires migrating all replicas each time we add or remove a single one. Our layout policy does not always maintain an optimal placement of replicas – placement is optimal only when the number of replicas is a power of two. However, the layout does perform well without requiring expensive migrations each time the number of replicas for a block changes. Note that for blocks with stable access patterns, where the number of replicas rarely changes, we evenly distribute replicas around the chain. Our layout is more resilient to failures than concentrated data layouts, since it spreads replicas evenly across nodes, rather than concentrating them on small subsets of nodes.

**Popularity-aware Replication and Reclamation.** Replication in DFSs is common to tolerate node failures and improve read throughput. Likewise, migrating popular replicas to active nodes is common in energy-efficient DFSs [48, 65, 129, 78]. BlinkFS also uses
replication to mitigate its latency penalty as power varies by employing popularity-aware replication and reclamation to reduce the latency for popular blocks. Note that our replication strategy is independent of the power level, since replicating at low power levels may be infeasible. In this case, a modest amount of battery-based storage may be necessary to spawn the appropriate replicas to satisfy performance demands [119]. By default, BlinkFS maintains three replicas per block, and uses any remaining capacity to potentially store additional latency-improving replicas.

As clients create new files or blocks become less popular, BlinkFS lazily reclaims replicas as needed. Using staggered blinking and a power-balanced data layout, the number of replicas $r$ required to ensure a block is available 100% of each blink interval, based on the total nodes $N$, blink interval $t$, available power $p$, and active node power consumption $p_{\text{node}}$, is $r = \lceil \frac{N}{(N-1)p_{\text{node}} - p} \rceil$. At low enough power levels, i.e., where $1 > \frac{p}{p_{\text{node}}}$, there are periods within each blink interval where no nodes are active. In this case, the minimum possible fraction of each blink interval the block is unavailable is $1 - \frac{p}{p_{\text{node}}}$, assuming it is replicated across all nodes.

The master uses the relationships above to compute a block’s access latency, given its replication factor and the current power level, assuming requests are uniformly distributed over each blink interval. There are many policies for spawning new replicas to satisfy application-specific latency requirements. In our prototype, the master tracks block popularity as an exponentially weighted moving average of a block’s I/O (read) accesses, updated by the proxy every blink interval, and replicates blocks every period in proportion to their relative popularity, such that all replicas consumes a pre-set fraction of the unused capacity. For frequently updated blocks, BlinkFS caps the replication factor at three, since excessive replicas increase write overhead. To replicate a block, the master selects a source and a destination block server based on blinking patterns of the nodes, and directs the source node to send the data to the destination node either directly – if both nodes are active at the same time – or via the proxy, otherwise.

### 6.5 Implementation

We implement a BlinkFS prototype in C, including a master (≈3000LOC), proxy (≈1000LOC), client (≈1200LOC), power manager (≈100LOC), power client (≈50LOC), and block server (≈900LOC). The client uses the FUSE (Filesystem in Userspace) library in Linux to transfer file system-related system calls from kernel space to user space. Thus, BlinkFS clients expose the POSIX file system API to applications. BlinkFS also extends the API with a few blink-specific calls, as shown in Table 1. These system calls enable applications to inspect information about node blinking patterns to improve their data access patterns, job scheduling algorithms, etc., if necessary. All other BlinkFS components run in user space. While the master, proxy, and power manager are functionally separate and communicate via event-based APIs (using libevent), our prototype executes them on the same node. To experiment with a wide range of unmodified applications, we chose to implement our prototype in FUSE, rather than extend an existing file system implementation, such as HDFS.

Our prototype includes a full implementation of BlinkFS, including the staggered blinking policy, power-balanced data layout, and popularity-aware replication. Our current im-
Table 6.1. POSIX-compliant API for BlinkFS

<table>
<thead>
<tr>
<th>FUSE Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>getattr(path, struct stat *)</td>
</tr>
<tr>
<td>mkdir(path, mode)</td>
</tr>
<tr>
<td>rmdir(path)</td>
</tr>
<tr>
<td>rename(path, newpath)</td>
</tr>
<tr>
<td>chmod(path, mode)</td>
</tr>
<tr>
<td>chown(path, uid, gid)</td>
</tr>
<tr>
<td>truncate(path, offset)</td>
</tr>
<tr>
<td>open(path, struct fuse_file_info *)</td>
</tr>
<tr>
<td>read(path, buff, size, offset, fuse_file_info*)</td>
</tr>
<tr>
<td>write(path, buff, size, offset, fuse_file_info*)</td>
</tr>
<tr>
<td>release(path, fuse_file_info*)</td>
</tr>
<tr>
<td>create(path, mode, fuse_file_info*)</td>
</tr>
<tr>
<td>fgetattr(path, stat*, fuse_file_info*)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BlinkFS-specific Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBlinkState(int nodeid)</td>
</tr>
<tr>
<td>getBlockInfo(int blockid)</td>
</tr>
<tr>
<td>getFileInfo(path)</td>
</tr>
<tr>
<td>getServerLoadStats(int nodeid)</td>
</tr>
</tbody>
</table>

Implementation redirects all writes through the proxy, but permits clients to issue reads directly to block servers if both are concurrently active. Also, we maintain an in-memory log of writes in the proxy, but currently do not mirror it to a backup. We implement the migration-based approach and Rabbit from Section 6.2 to compare with BlinkFS. We also compare the load-proportional blinking policy from Chapter 5, which blinks nodes in proportion to the popularity of blocks they store. The policy is useful for access patterns with skewed popularity distributions, e.g., Zipf, and does not require migrations.

**Hardware Prototype.** We construct a small-scale hardware prototype (Figure 6.6) that uses intermittent power to experiment with BlinkFS in a realistic setting. We use a small cluster of ten Mac minis running Linux kernel 2.6.38 with 2.4Ghz Intel Core 2 Duo processors and 2GB of RAM connected together using an energy-efficient switch (Netgear GS116) that consumes 15W. Each Mac mini uses a flash-based SSD with a 40GB capacity. We also use a separate server to experiment with external always-on clients, not co-located with block servers. To minimize S3 transition times, we boot each Mac mini in text mode, and unload all unnecessary drivers. With the optimizations, the time to transition to and from ACPI’s S3 state on the Mac mini is one second. Note that much faster sleep transition times, as low as a few milliseconds, are possible, and would further improve BlinkFS’s performance. Unfortunately, manufacturers do not optimize sleep transition time in today’s server-class nodes. Fast millisecond-scale transitions, as in PowerNap [127], significantly improve performance, especially access latency, by reducing the blink interval’s length, but are not yet commercially available in today’s servers.

We select a blink interval of one minute, resulting in a transition overhead of $\frac{1}{60}=1.67\%$ every blink interval. We measure the power of the Mac mini in S3 to be 1W and the power in S0 to be 25W. Thus, in S3, nodes operate at 4% peak power. Since BlinkFS requires at least one node (to host the master, proxy, and power manager) and the switch to be active, its minimum power consumption is 40W, or 15% of its maximum power consumption. The remaining nine nodes each run a power client, block server, and BlinkFS client. We power the cluster from a battery that connects to four ExTech 382280 programmable power
supplies, each capable of producing 80W, that replay the variable power traces below. To prevent the batteries from over and under-charging we connect the energy source to the battery using a TriStar T-60 charge controller. We also use two DC current transducers and a voltage logger from National Instruments to measure the current flowing in and out of the battery and the battery voltage, respectively. Our experiments use the battery as a short-term buffer of five minutes; optimizations that utilize substantial battery-based storage are outside the scope of this thesis.

**Power Signals.** We program our power supplies to replay DC generation traces from our own small-scale solar panel and wind turbine deployment and 2) traces based on wholesale electricity prices. We also experiment with both multiple steady and oscillating power levels as a percentage, where 0% oscillation holds power steady throughout the experiment and $N\%$ oscillation varies power between $(45 + 0.45N)\%$ and $(45 - 0.45N)\%$ every five minutes.

For our renewable trace, we combine traces from our solar/wind deployment, and set a minimum power level equal to the power necessary to operate BlinkFS’s switch and master
node (40W). We compress our renewable power signal to execute three days in three hours, and scale the average power to 50% of the cluster’s maximum power. Note that the 24X compressed power signal is not unfair to the migration-based approach, since our data sets are relatively small (less than 20GB). We would expect large clusters to store more than 24X this much data, increasing the relative transfer time for migration. BlinkFS’s performance is, by design, not dependent on the data set size. For our market-based electricity price trace, we use the New England ISO 5-minute spot price of energy for the 3-hour period from 7am to 10am on September 22, 2011, assuming a fixed monetary budget of 1¢/kWh; ISO’s regulate wholesale electricity markets in the U.S. The average price in the trace is 4.5¢/kWh, the peak price is 5.2¢/kWh, and the minimum price is 3.5¢/kWh. We envision utilities increasing their use of market-based electricity pricing in the future, as electricity demands and prices increase. For example, Illinois already requires utilities to provide residential customers with access to hourly market-based electricity prices based directly on the wholesale price [107].

6.6 Evaluation

We first benchmark BlinkFS’s overheads as a baseline for understanding its performance at different steady and oscillating power levels. We then evaluate BlinkFS for three different applications: a MapReduce-style application [115] (a data-intensive batch system), unmodified MemcacheDB [108] (a latency-sensitive key-value store), and file system traces from a search engine. Each application runs as a normal process with access to the BlinkFS mount point.

6.6.1 Benchmarks

To benchmark BlinkFS, we wrote a single-threaded application that issues blocking read/write requests to the client’s interface, rather than through FUSE, to examine performance independent of FUSE overheads. One limitation of FUSE is that the maximum size of write and read requests are 4KB and 128KB, respectively, irrespective of BlinkFS’s block size. The breakdown of the latency overhead at each component for a sample 128KB
read is 2.5ms at the proxy, 0.57ms at the block server, 2.7ms at the client, and 0.33ms within FUSE for a total of 6.1ms. The results demonstrate that BlinkFS’s overheads are modest. We also benchmark BlinkFS’s maximum sequential read and write throughput (for a single replica) at full power for a range of block sizes. Figure 6.7 shows that, as expected, read and write throughput increase with increasing block size. However, once block size exceeds 4MB throughput improvements diminish, indicating that I/O transfer overheads begin to dominate processing overheads.

Read and write throughput via the proxy differ because clients off-load writes to proxies, which return before applying the writes to block servers. We also benchmark the throughput for reads sent directly to the proxy, which shows how much the proxy decreases maximum throughput (∼40% for large block sizes). The overhead motivates our client optimization that issues reads directly to the block server, assuming both are concurrently active. The throughput of writes sent directly to block servers is similar to that of reads. We ran a similar experiment using 4MB blocks that scales the number of block servers, such that each block server continuously receives a stream of random I/O requests from multiple clients (using a block size of 4MB). As shown in Figure 6.9, write throughput reaches its maximum using three block servers, due to CPU overheads. The result shows that in the worst case a proxy-to-block server ratio larger than 1:3 does not improve write throughput; for realistic workloads, each proxy is capable of supporting at least ten nodes, as our case studies demonstrate.

We also benchmark the read and write latency for different block replication factors for a range of power levels. Figure 6.8(a) shows that average read latency increases rapidly when using one replica if available power drops below 50%, increasing to more than 8 seconds. Additional replicas significantly reduce the latency using staggered blinking: in our prototype, all blocks are always available, i.e., stored on an active node, when using six replicas at 20% power. Write latency exhibits worse performance as we increase the number of replicas. In this benchmark, where clients issue writes as fast as possible, the proxy must apply writes to all replicas, since its log of pending writes becomes full (Figure 6.8(b)). Since the increase in the write latency is much less than the increase in read latency, the trade-off is acceptable for workloads that mostly read data. Table 6.2 shows the

Table 6.2. Standard deviation and 90th percentile latency.

<table>
<thead>
<tr>
<th>Replication factor</th>
<th>Std Dev</th>
<th>90\textsuperscript{th} per</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>1619</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>15524</td>
</tr>
<tr>
<td>3</td>
<td>W</td>
<td>6017</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>5476</td>
</tr>
<tr>
<td>6</td>
<td>W</td>
<td>8883</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>523</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Latency (ms)</th>
<th>Power (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.6</td>
</tr>
<tr>
<td>40</td>
<td>0.7</td>
</tr>
<tr>
<td>60</td>
<td>0.8</td>
</tr>
<tr>
<td>80</td>
<td>0.9</td>
</tr>
<tr>
<td>100</td>
<td>1.0</td>
</tr>
</tbody>
</table>
standard deviation and 90th percentile latency for read and write requests as the replication factor and power levels change.

Finally, we benchmark the overhead to migrate data as power oscillates, to show that significant data migration is not appropriate for intermittent power. For the benchmark, we implement a migration-based approach that equally distributes data across the active nodes. As power varies, the number of active nodes also varies, forcing migrations to the new set of active nodes. We oscillate available power every five minutes, as described in Section 6.5. We wrote a simple application that issues random (and blocking) read requests; note that the migration-based approach does not respond to requests while it is migrating data. Figure 6.10 shows that read throughput remains nearly constant for BlinkFS at different oscillation levels, whereas throughput decreases for the migration-based approach as oscillations increase. Further, the size of the data significantly impacts the migration-based approach. At high oscillation levels, migrations for a 20GB data set result in zero effective throughput. For smaller data sets, e.g., 10GB, the migration-based approach performs slightly better than BlinkFS at low oscillation levels, since the overhead to migrate the data is less than the overheads associated with BlinkFS.

Small power variations trigger large migrations in large clusters. If a 1000 node cluster with 500GB/node varies power by 2% (the average change in hourly spot prices), it must deactivate 20 nodes, causing a 10 terabyte migration. Even with a 10 gigabit link, the migration would take >2 hours, and prevents the cluster from performing useful work. We compare the migration-based approach with BlinkFS for these small power variations.

6.6.2 Case Studies

We experiment with a MapReduce-style application, MemcacheDB, and file system traces from a search engine using the traces discussed in Section 6.5 for three different approaches: BlinkFS, Rabbit, and Load-proportional. Since MapReduce executes batch jobs, it is well-suited for intermittent power if its jobs are tolerant to delays. We also experiment with interactive applications (MemcacheDB and file system traces) to demonstrate BlinkFS’s flexibility. To fairly compare with Rabbit, we use an equal-work layout where the first two nodes store primary replicas, the next five nodes store secondary replicas, and the last two nodes store tertiary replicas.
Figure 6.10. BlinkFS performs well as power oscillation increases.

Figure 6.11. MapReduce completion time at steady power levels and using our combined wind/solar power trace.

Note that, while Rabbit performs well in some instances, it relies on severe capacity limitations, as described in Section 6.2, to avoid migrations. For BlinkFS’s power-balanced data layout, we use 2/9ths of the capacity to store one replica of each block, and the rest to store additional replicas. We set the default number of replicas to three, with a maximum replication factor of six for our popularity-aware replication policy. For load-proportional, we arrange blocks on nodes a priori based on popularity (from an initial run of the application) to eliminate data migrations, which provides an upper bound on load-proportional performance. Since MapReduce co-locates computation and data, the nodes execute both a client and a block server. For the other applications, we use an external, e.g., always-on, client. Finally, we use a 4MB block size.

MapReduce. For MapReduce, we create a data set based on the top 100 e-books over the last 30 days from Project Gutenberg (http://www.gutenberg.org/). We randomly merge these books to create 27 files between 100 and 200MB, and store them in our file system. We then write a small MapReduce scheduler in Python, based on BashReduce, that partitions the files into groups for each job, and sends each group to a MapReduce worker node, co-located on each block server. We execute the simple WordCount MapReduce application, which reads files on each node, counts the words in those files, and sends the results back to the scheduler. The scheduler then executes a reduce step to output a file containing all distinct words and their frequency.
We experiment with MapReduce using both constant and intermittent power. At constant power, Figure 6.11(a) shows that the completion time is nearly equal for all three policies at high power, but BlinkFS outperforms the others at both medium and low power. For instance, at 50% power BlinkFS improves completion time by 42% compared with Rabbit and 65% compared with load-proportional. Note that at low power (20%), MapReduce stalls indefinitely using Rabbit, since it requires at least two active nodes to ensure all data is accessible. Both Rabbit and Load-proportional also impact MapReduce computations by deactivating or reducing, respectively, the active time of cluster nodes as power decreases. BlinkFS does not affect the scheduling or placement policy as power varies.

For variable power, we execute a stream of smaller jobs, which process data sets that only consist of 27 e-books, to track the number of jobs we complete every five minutes. For this experiment, Figure 6.11(b) shows that BlinkFS outperforms Load-proportional at all power levels, since it does not skew the active periods of each node. While Rabbit performs better at high power levels, it stalls indefinitely whenever power is unable to keep all data accessible, i.e., two active nodes.

MemcacheDB Key-Value Store. MemcacheDB is a persistent version of memcached, a widely-used distributed key-value store for large clusters. MemcacheDB uses BerkeleyDB as its backend to store data. We installed MemcacheDB on our external node, and configured it to use BlinkFS to store its BerkeleyDB. To avoid any caching effects, we configure MemcacheDB to use only 128 MB of RAM and set all caching-related configuration options to their minimum possible value. We then populated the DB with 10,000 100KB objects, and wrote a MemcacheDB workload generator to issue key requests at a steady rate according to a Zipf distribution.

Our results show that BlinkFS and Rabbit perform well at high and medium constant power levels (Figure 6.12(a)), while load-proportional performs slightly worse. Load-proportional does not benefit from replication, since replicas of popular blocks are inevitably stored on unpopular nodes. Thus, BlinkFS significantly outperforms load-proportional at low power levels. As with MapReduce, Rabbit has infinite latency at low power, since its data is inaccessible. Next, we run the same experiment using our wind/solar signal and observe the average request latency over each 5-minutes interval. As shown in Figure 6.12(b), BlinkFS performs better than load-proportional at nearly all power levels. The latency for
BlinkFS scales up and down gracefully with the power signal. As in the MapReduce example, Rabbit performs better, except when available power is not sufficient to keep primaries active.

**Search Engine.** We emulate a search engine by replaying file system traces, and measuring the number of queries serviced each minute. To emulate the trace, we created a 30GB file divided into 491,520 blocks of size 64KB and implemented an emulator in Python to issue I/O requests. We run the experiment with the power signals described in Section 6.5. As Figure 6.13 shows, BlinkFS outperforms load-proportional at all power levels. As expected, the migration-based approach performs slightly better than BlinkFS at steady power levels, but much worse for even slight (∼10%) fluctuations in the available power. Since the available power is always more than the power required to run two nodes, Rabbit (not shown for clarity), outperforms (57 queries/minute) the others, since it stores primary replicas on these two nodes. Even for such a small dataset and power fluctuations, BlinkFS satisfies 12% and 55% more requests within a 3-hour period than the migration-based and load-proportional approaches.

### 6.7 Conclusion

This chapter presents the design of BlinkFS to handle significant, frequent, and unpredictable changes in available power. Our design includes techniques to mitigate blinking’s latency penalty. Recent work [120] also highlights the difficulty of designing latency-sensitive applications that are also energy-efficient. Intermittent power enables opportunities for optimizing data centers to be cheaper and greener, including incorporating more intermittent renewables, leveraging market-based electricity pricing, operating during extended blackouts, and fully utilizing a data center’s power delivery infrastructure. Regulating power usage independent of workload demands enables data centers to leverage these optimizations. While today’s energy prices do not strongly motivate intermittent power optimizations, such as increasing the fraction of renewable power in data centers, companies remain interested to be environmentally friendly and due to expectations of future increases in energy prices. We envision rising energy prices to incentivize data centers to design systems optimized for intermittent power in the future.
CHAPTER 7
MULTIMEDIA CACHE FOR INTERMITTENT POWER

We have seen how a distributed in-memory cache could be redesigned to perform well on intermittent power. But, unlike an in-memory cache, multimedia caches maintain persistent states which become unavailable at low power. Furthermore, maintaining states availability becomes challenging if the available power changes frequently and significantly. In this chapter I design a distributed multimedia cache for intermittent power, which combines a staggered load-proportional blinking with 1st chunk replication to minimize bandwidth cost while reducing buffering time at all power levels.

7.1 Background and Motivation

Multimedia caches are used by content providers for storing contents near end users to reduce backhaul bandwidth usage and buffering time. Similar to other caches, a multimedia cache is also used for performance optimization, and not for application correctness. Traditional multimedia caches assume unlimited supply of power and focus primarily on reducing backhaul bandwidth usage. But running a multimedia cache on intermittent power warrants new approaches as a period of low power makes cached contents unavailable, in which case it can either forward a request to back-end servers or wait for the cached contents to become available. Sending a request for the cached data to back-end servers increases the bandwidth cost, whereas waiting for the cached data to become available increases the access latency. Minimizing both the bandwidth cost and buffering time necessitates new design techniques that optimize performance while dealing with intermittent power constraints.

There could be a number of scenarios where we need to run a multimedia cache on intermittent power. One possible use case is to use a multimedia cache to augment off-the-grid cellular towers. Cellular networks in developing countries have historically had low reliability, since cellular towers typically run off power from the electric grid. As the recent Indian blackout in July 2012, which left 700 million people without power for two days, indicates, the grid in these countries is often unstable. In some cases, connections to an even unstable electric grid may not be available. For example, in India, 150,000 out of 400,000 cell towers do not have reliable access to the electric grid [89]. Grid instability and lack of infrastructure has led network operators to run cellular towers “off the grid” [86]. Today’s “off the grid” cellular towers operate off diesel generators that are costly to maintain, in large part, because operators must continuously refuel them with expensive and “dirty” diesel fuel. Since i) network operators view developing countries as a large potential market and ii) many potential users in these countries reside in rural areas without access to a
reliable electric grid, there is a strong financial incentive to deploy off-the-grid cell towers powered by renewable energy sources, e.g., from either solar panels or wind turbines.

In parallel, the growth of smartphones as a primary end-point for multimedia data has led to a significant rise in bandwidth usage of cell towers, which requires higher backhaul bandwidth to fetch data, as well as additional spectrum or denser tower deployments to deliver this data. As a result, in addition to the problems above, network operators have also sought ways to reduce the bandwidth consumption of multimedia applications. Traditionally, network operators have deployed caches only at centralized locations, such as operator peering points, in part, for both simplicity and ease of management [102]. However, researchers and startups have recently recognized the benefit of placing caches closer to edge [102, 84]. Co-locating server caches closer to cell towers would both reduce access latency, by eliminating the need to route requests for cached data through a distant peering point, and reduce backhaul bandwidth usage from the cell tower to the peering point. Caches co-located with cell towers primarily target multimedia data, since it consumes the largest fraction of bandwidth and is latency-sensitive.

In this chapter we consider how to operate a distributed cache for multimedia data that is co-located with a cell tower powered using renewable energy. A key challenge in using renewable sources such as solar and wind is that they are intermittent and the amount of power they generate fluctuates depending on environmental conditions. Consequently, we assume that when renewable generation is plentiful, e.g., on sunny or windy days, the generated power is sufficient to power both the cell tower and the co-located servers that house the multimedia cache. However, during periods of scarcity the renewable sources (combined with a modest-sized battery) may not generate sufficient power and the setup must reduce its total energy footprint. In this case, we assume that the available scarce power is used to keep the cell tower up and running (so that consumers do not lose cellular service) and that there may not be enough energy to always operate the multimedia cache.

In Chapter 5 we showed that blinking improves the performance of an in-memory cache for small objects, e.g., Memcached storing simple strings and binary values. Caches for multimedia data differ in important ways. First, multimedia contents are often much larger in size than Memcached objects, which are limited to 1MB in size, and thus require persistent storage for caching. Second, unlike Memcached objects, multimedia data can be streamed, such that a video need not be completely transmitted to a client before the client can display it. Instead, the client can start playing a video as soon as it receives data for the first few seconds, as long as the client continues streaming data from the server while the video is playing. This characteristic is well-suited to the blinking abstraction, since different chunks of a video can be stored on different cache servers in a cluster. The challenge is to activate nodes in a way that segments of the video are streamed to the client in a sequence that allows for uninterrupted video playback.

In this chapter we design GreenCache, a distributed multimedia cache for intermittent power. For the sake of presentation, we assume that GreenCache is co-located with cell towers that run off renewable energy. GreenCache leverages the blinking abstraction to significantly (i) modulate its energy footprint to match available power, (ii) reduce bandwidth usage from the cell tower to backend servers, and (iii) reduce access latency for clients despite fluctuations in available power. As discussed above, minimizing bandwidth usage (or cost) and maximizing users’ experience, e.g., by reducing buffering time, are two
primary goals of a multimedia cache. We analyze video traffic behavior of a large number of users for the most popular user-generated video site, YouTube, and exploit traffic characteristics and video properties to design new placement and blinking policies for minimizing bandwidth usage and maximizing users’ experience. In achieving this goal, this chapter makes the following contributions:

**YouTube Trace Analysis.** To motivate the design and feasibility of GreenCache and the use of the blinking abstraction in streaming environments, we collect YouTube video traces from the University of Connecticut for 3 days, and analyze them to find important characteristics about viewers behavior and the popularity of YouTube videos.

**GreenCache Design.** We detail the design of a blink-aware distributed multimedia cache and its advantages over existing multimedia caches when using renewable energy sources. Our design uses an always-active proxy to receive requests, while masking blinking, from mobile clients.

**Buffering Time Reduction Techniques.** We detail techniques for reducing video buffering time at the client. Our approach combines a load-proportional data layout with a staggered load-proportional blinking policy to minimize per video average buffering time. Our design also uses a popularity-aware cache eviction policy to minimize bandwidth usage from cell towers to backend servers.

**GreenCache Implementation and Evaluation.** We implement GreenCache on a laboratory prototype comprising a real 4G WiMAX base-station we have deployed on the UMass campus and a cluster of ten Mac Minis that can be powered using either a renewable source or a programmable power supply that mimics these renewables. We then evaluate our cache for bandwidth usage and buffering time at different (fixed and oscillating) power levels using our WiMAX base-station and WiMAX clients. Our experimental workloads are based on user requests from the real network traces mentioned above. Our results show that a staggered load-proportional blinking policy, which staggers when nodes are active and varies the length of the active interval, results in 3X less buffering (or pause) time by the client compared to an activation blinking policy, which simply activates and deactivates nodes as power fluctuates, for realistic power variations from renewable energy sources.

The remainder of the chapter is outlined as follows. Section 7.2 describes the benefits of blinking for caches running on intermittent power. Section 7.3 analyzes YouTube video traces from the University of Connecticut, while Section 7.4 presents design techniques for a blinking multimedia cache. Section 7.6 details our implementation of a small distributed cache prototype, and Section 7.6 then evaluates our prototype for two metrics – bandwidth cost and buffering time. Finally, Section 7.7 discusses related work and Section 7.8 concludes.

### 7.2 Cache and Intermittent Power

Our work assumes cellular towers that are powered using off-the-grid sources such as solar and wind energy. We assume that the cellular network employs in-network multimedia caches to deliver popular content to mobile end-users via cellular data connections. To reduce latency, backend server load, as well as backhaul/upstream bandwidth, we assume that these caches are located close to, or at, the cellular towers of 3G/4G base stations.
As noted in Section 7.1, while today’s legacy cellular networks are not able to co-locate caches very close to cell towers due to inherent architectural limitations, research efforts and startup companies [102, 84] are developing techniques and products to address these limitations; thus we envision that future cellular networks will be able to deploy computation and storage, e.g., server caches, near base stations—similar to what we assume here.

The use of caches in a cellular network has many benefits. Since the popularity distribution of videos is often heavy-tailed, i.e., a small set of videos out of the pool of all available videos are significantly more popular than the rest, a well-designed cache cluster can reduce the back-end traffic, and thus the bandwidth cost on the uplink. Our earlier work analyzed the benefits of caching for YouTube videos through trace-based simulations, and showed that caching can indeed reduce uplink bandwidth [103, 90]. Further, an increase in storage capacity increases the efficiency of the cache. i.e., the larger the cache’s storage the higher the potential that a requested video can be served from the cache.

However, co-locating caches near cellular towers also raises challenges. First, the presence of servers and storage near the cellular tower increases the energy footprint of the tower. The problem is exacerbated in developing countries with an unreliable grid. In off-the-grid towers with renewable sources, we must deal with the additional problem intermittency in these energy sources. Figure 7.1 shows how both solar and wind energy can vary each day. The figure shows that even on generally sunny or windy days the output from renewables can fluctuate significantly. Even with the use of batteries, there may not be sufficient energy to operate the base station and the servers during periods of energy scarcity (e.g., on cloudy days with low solar output). We assume that the server caches must somehow reduce their energy usage during such periods while the base station stays up—to the extent possible—to provide cellular service to end users.

To handle energy scarcity, we assume a cache architecture that comprises of a number of low-power servers, since a single large cache is not well-suited to operating off intermittent renewable energy sources. To understand why, consider that since computing equipment, including servers and network switches, is not energy-proportional, it is not possible to scale down performance with power usage to adapt to changes in available power. For example, a 300W server may have a dynamic power range when active between 200W

Figure 7.1. Solar and wind energy harvesting from our solar panel and wind turbine deployment on three consecutive days in Sep 2009.
and 300W; thus, if power generation drops below 200W the server must be shutdown. To
mimic energy-proportionality, an alternative approach uses smaller, lower-power servers,
which can each be activated and deactivated to match available power [99]. The advantage
of this approach is that it allows the cache size to scale up and down based on available
power. However, it introduces a new complication: if servers are inactive due to power
shortages by renewables then the data cached on them becomes unavailable. If data resides
on an inactive server, the client must either wait until the server is active, e.g., there is
enough power, or retrieve the already cached data again from the origin server.

With a distributed cache, there are two ways to reduce energy use during shortfall pe-
riods. First, some or all caches can be temporarily powered down, but doing so implies
that users will not see any benefits of caching in these periods. A better approach is to use
blinking [93] where each node rapidly transitions (duty cycles) between sleep and awake
modes. Blinking allows caches to provide service, albeit at degraded performance, during
shortfall periods. Our earlier work demonstrated the feasibility of implementing blinking
on commodity hardware—the rate and duration of blinking can be adjusted to match the
energy availability. Longer sleeps can be used, for instance, during very low energy supply
periods.

In essence, blinking provides a cache with new options in its design space. Rather than
having a small cache composed of the number of always-on servers the available power can
sustain, blinking provides the option of having a much larger cache composed of servers
that are active for only a fraction of time each blink interval, e.g., active for 10 seconds
during each minute interval. The use of blinking raises new challenges in multimedia
cache design. The main challenge is to ensure smooth uninterrupted video playback even
while blinking. Doing so implies that caches have to stream additional data during their
active periods to compensate for lack of network streaming during sleep periods. Further,
end-clients will need to employ additional client-side buffers and might see higher startup
latencies.

Since multimedia applications are very sensitive to fluctuation in network bandwidth
that might cause delayed data delivery at the client, most applications like video players
employ a buffer to smooth out such fluctuations and provide an uninterrupted, error free
play out of the data. This buffer, which already exists for most multimedia applications on
the client side, integrates well into the blinking approach since it also allows the cache to
bridge outage times in individual cache servers, as shown in Figure 7.2. A blinking cache
will stream additional chunks when active, which are buffered at the client. As shown in
this figure, the player is then able to play the video smoothly and masks interruptions from
the viewer as long as it gets the next chunk of data before the previous chunk has finished
playing.

Finally, in a typical cell tower or 3G/4G/WiMAX scenario the downstream bandwidth
(~30-40 Mbps) is much less than the bandwidth a cache server can provide, which is
generally limited by its network card and disk I/O. So, the cache server can potentially
reduce its energy consumption by sending data at its full capacity for a fraction of a time
interval (usually few seconds) and going to a low-power state for the remaining period of
the time interval, as shown in Figure 7.2. In essence, the server could employ the blinking
abstraction to reduce its energy footprint while still satisfying the downstream bandwidth
requirement of the cell tower or WiMAX station. Moreover, blinking facilitates a cache to
employ more servers than it can keep active with the available power, and thus provides an opportunity to reduce server load and bandwidth usage.

The primary drawback of a blinking cache is that it stalls a request if the requested video is not currently available on an active server. If a client requests a video that is present on an inactive server, the cache can either get the video from the back-end server or the client pauses play out until the inactive server becomes active. While getting the video from the back-end server, instead of waiting for the inactive server to become active, reduces the buffering time, it increases the bandwidth cost. As described in Section 7.4, GreenCache uses a low-power always-on proxy and staggered load-proportional blinking policy to reduce buffering time while sending requests to back-end servers only if data is not available in the cache.

7.3 GreenCache Feasibility: Trace Analysis

To inform the design of GreenCache based on the characteristics of multimedia traffic and viewer behavior, we analyze a network trace that was obtained by monitoring YouTube traffic entering and leaving a campus network at the University of Connecticut. We believe that such a trace can be seen as a first order approximation for a community that is served by a WiMAX or 3G/4G base station. (One can easily imagine that a University campus or a substantial part of the campus could be served by such a base station.) Indeed, some Universities have started installing WiMAX base stations on their campuses as part of their network infrastructure [91].

The network trace is collected with the aid of a monitoring device consisting of PC with a Data Acquisition and Generation (DAG) card [83], which can capture Ethernet frames. The device is located at a campus network gateway, which allows it to capture all traffic to and from the campus network. It was configured to capture a fixed length header of all HTTP packets going to and coming from the YouTube domain. The monitoring period for this trace was 72 hours. This trace contains a total of 105,339 requests for YouTube videos.
out of which $\sim 80\%$ of the video requests are single requests which leaves about 20\% of the multiple requests to take advantage of caching of the requested videos. We would like to point out that a similar caching potential (24\% in this case) has been reported in a more global study of 3G networks traffic analysis by Erman et al. [88].

Figure 7.3 shows the popularity distribution of the 100 most popular videos, which is obtained based on the user requests recorded in the trace. This figure only shows the 100 most popular videos since the trace contains many videos with a very low popularity (< 10 requests) and we wanted to depict the distribution of the most popular videos in more detail. The data obtained from the analysis of the trace shows that, despite the very long tail popularity distribution, caching can have an impact on the performance of such a video distribution system.

In earlier work [90], we have shown that, not only caching but also the prefetching of prefixes of videos that are shown on the related video list of a YouTube video page can improve the viewers experience of watching videos. Further analysis of the trace revealed that 47,986 request out of the 105,339 had the tag related_video ($\sim 45\%$), which indicates that these videos have been chosen by viewers from the related video list that is shown on each YouTube video’s web page. In addition to identifying videos that are selected from the related list, we also determine the position on the related list the video was selected from and show the result in Figure 7.4. It shows that users tend to request from the top 10 videos shown on the related list of a video, which accounts for 80\% of the related video requests in the trace. This data shows that, prefetching the prefixes of the top 10 videos shown on the related list of a currently watched video can significantly increase viewer’s experience, since the initial part can be streamed immediately from a location close to the client. Based on these results, we decided to evaluate a blinking multimedia cache that performs both, traditional caching, and prefix prefetching for the top 10 videos on the related video list.

We also analyze the trace to investigate if viewers switch to a new video before they completely finish watching the current video. In order to analyze this behaviour, we look into the timestamps of a user requesting two consecutive videos. We calculate the difference of these timestamps and compare it with the total length of the first video requested to determine if the user has switched between videos before the previous video is completely viewed.

Figure 7.5 shows the number of occurrences (in percent out of the total number of videos watched) a video is watched for $x\%$ of its total length. This result shows that only in
45% of the cases videos are watched completely (also this number is similar to the global study performed by Erman et al. [88]). In all other cases only part of the video is watched, with the majority of these cases (∼ 40%) falling in the 0 - 20% viewing session length. This result let us to the decision to divide a video into equal-sized chunks, which allows for the storage of different chunks that belong to a single video on different nodes of the cache cluster. In Section 7.4.1, we describe how the chunk size is determined and how chunking a video can reduce the uplink bandwidth usage if used on a blinking multimedia cache cluster.

7.4 GreenCache Design

Figure 7.6 depicts GreenCache’s architecture, which consists of a proxy and several cache servers. The proxy maintains a video→chunk mapping and a chunk→node mapping, while also controlling chunk server placement and eviction. Clients, e.g., web browsers on smartphones, connect to video servers through the proxy, which fetches the requisite data from one or more of its cache servers, if the data is resident in the cache. If the data is not resident, the proxy forwards the request to the host, i.e., backend server. The proxy stores metadata to access the cache in its own memory, while video chunks reside on stable storage on each cache server.

GreenCache also includes a power manager, that monitors available power and energy stored in a battery using hardware sensors, e.g., a voltage logger and current transducer. The power manager implements various blinking policies to control nodes’ active and inactive
intervals to match the cache’s power usage to the available power. The power manager communicates with a power client running on each cache server to set the start time and active period every blink interval. The power client activates the node at the start time and deactivates the node after the active period every blink interval, and thus controls node-level power usage by transitioning the node between active and inactive states.

As discussed earlier, the primary objective of multimedia cache is to reduce buffering (or pause) time at the client and the bandwidth usage between the cache and the origin server. Next, we describe GreenCache’s techniques to both reduce bandwidth usage to the backend origin server, while also minimizing buffering (or pause) time at the client.

### 7.4.1 Minimizing Bandwidth Cost

As Figure 7.3 indicates, all videos are not equally popular. Instead, a small number of videos exhibit a significantly higher popularity than others. Similar to other multimedia caches, GreenCache has limited storage capacity, requiring it to evict older videos to
cache new videos. An eviction strategy that minimizes the bandwidth usage each interval will evict the least popular videos during the next interval. However, such a strategy is only possible if the cache knows the popularity of each video in advance. To approximate a video’s future popularity, GreenCache maintains each video’s popularity as an exponentially-weighted moving average of a video’s accesses, updated every blink interval. The cache then evicts the least popular videos if it requires space to store new videos.

As shown in Figure 7.5, most videos are not watched completely most of the time. In fact, the figure shows that users of YouTube watch less than 45% of the videos to completion. In addition, users might watch the last half of a popular video less often than the first half of an unpopular video. To account for discrepancies in the popularity of different segments of a video, GreenCache divides a video into multiple chunks, where each chunk’s playtime is equal in length to the blink interval. Similar to entire videos, GreenCache tracks chunk-level popularity as an exponentially weighted moving average of a chunk’s accesses. Formally, we can express the popularity of the $i_{th}$ chunk after the $t_{th}$ interval as:

$$Popularity_i^t = \alpha A_i^t + (1 - \alpha)Popularity_i^{t-1}$$  \hspace{1cm} (7.1)

$A_i^t$ represents the total number of accesses of the $i_{th}$ chunk in the $t_{th}$ interval, and $\alpha$ is a configurable parameter that weights the impact of past accesses. Further, GreenCache manages videos at the chunk level, and evicts least popular chunks, from potentially different videos, to store a new chunk. As a result, GreenCache does not need to request chunks from the backend origin servers if the chunk is cached at one or more cache servers.

### 7.4.2 Reducing Buffering Time

As discussed earlier, blinking increases buffering time up to a blink interval, if the requested chunk is not present on an active server. The proxy could mask the buffering time from a client if the client receives a chunk before it has finished playing the previous chunk. Assuming sufficient energy and bandwidth, the proxy can get a cached chunk from a cache server within a blink interval, since all servers become active for a period during each blink interval. As a result, a user will not experience pauses or buffering while watching a video in sequence, since the proxy has enough time to send subsequent chunks (after the first chunk) either from the cache or the origin server before the previous chunk finishes playing, e.g., within a blink interval. However, the initial buffering time for the first chunk could be as long as an entire blink interval, since a request could arrive just after the cache server storing the first chunk becomes inactive. Thus, to reduce the initial buffering time for a video, the proxy replicates the first chunk of cached videos on all cache servers. However, replication alone does not reduce the buffering time if all servers blink synchronously, i.e., become active at the same time every blink interval. As a result, as discussed next, GreenCache employs a staggered load-proportional blinking policy to maximize the probability of at least one cache server being active at any power level.

#### 7.4.2.1 Staggered Load-Proportional Blinking

As discussed above, we replicate the first chunk of each cached video on all cache servers in order to reduce initial buffering time. To minimize the overlap in node active in-
tervals and maximize the probability of at least one active node at all power levels, Green-Cache staggers start times of all nodes across each blink interval. Thus, every blink interval, e.g., 60 seconds, each server is active for a different period of time, as well as a different duration (discussed below). At any instant, a different set of servers (and their cached data) is available for clients. Since at low power the proxy might not be able to buffer all subsequent chunks from blinking nodes, clients might face delays or buffering while watching videos (after initially starting them).

To reduce the intermediate buffering for popular videos, GreenCache also groups popular chunks together and assigns more power to nodes storing popular chunks than nodes storing unpopular chunks. Thus, nodes storing popular chunks are active for a longer duration each blink interval. GreenCache ranks all servers from 1...N, with 1 being the most popular and N being the least popular node. The proxy monitors chunk popularity and migrates chunks to servers in rank order. Furthermore, the proxy distributes the available power to nodes in proportion to the aggregate popularity of their chunks. Formally, active period for the $i_{th}$ node, assuming negligible power for inactive state, could be expressed as

$$Active_i = \frac{BI \times P \times Popularity_i}{MP \times \sum_{k=1}^{n} Popularity_i}$$

(7.2)

$BI$ represents the length of a blink interval, $Popularity_i$ represents the aggregate popularity of all chunks mapped on the $i_{th}$ node, $P$ denotes the available power, and $MP$ is the maximum power required by an active node. Additionally, start times of nodes are staggered in a way that minimizes the unavailability of first chunks, i.e., minimizes the period when none of the nodes are active, every blink interval. Figure 7.7 depicts an example of staggered load-proportional blinking for five nodes. Note that since the staggered load-proportional policy assigns active intervals in proportion to servers’ popularity, it does not create an unbalanced load on the cache servers.

7.4.2.2 Prefetching Recommended Videos

Most popular video sites display a recommended list of videos to users. For instance, YouTube recommends a list of twenty videos which generally depends on the current video being watched, the user’s location, and other factors including past viewing history. From the trace analysis provided in Section 7.3, we can infer that, users tend to select the next video from recommended videos ~45% of the time. In addition, a user selects a video at the top more often than a video further down in the recommended list. In fact, Figure 7.4 shows that nearly 55% of the time a user selects the next video from top five videos in the recommended list. To further reduce initial buffering time the proxy prefetches the first chunk of top five videos in the recommended list, if these chunks are not already present in the cache. The proxy fetches subsequent chunks of the video when the user requests the video next.
7.5 GreenCache Implementation

We implement a GreenCache prototype in Java, including a proxy (~1500 LOC), cache server (~500 LOC), power manager (~200 LOC), and power client (~150 LOC). Mobile clients connect to the Internet through a wireless base station, such as a cell tower or WiMAX base station, which is configured to route all multimedia requests to the proxy. While the power manager and proxy are functionally separate and communicate via well-defined APIs, our prototype runs both modules on the same node. The power manager exposes APIs to access the available energy, blink interval, and node’s blink state – start time and active period. Our prototype does not require any modification in the base station or mobile clients. Both cache server and power client run together on each blinking node.

Our prototype includes a full implementation of GreenCache, including the staggered load-proportional blinking policy, load-proportional chunk layout, prefetching, video chunking, chunk eviction and chunk migration. The proxy uses a Java Hashtable to map videos to chunks and their locations, e.g., via their IP address, and maintains their status, e.g., present or evicted. Since our prototype has a modular implementation, we are able to experiment with other blinking policies and chunk layouts. We implement the activation and proportional policies from the original Blink work [93] to compare with GreenCache’s staggered load-proportional policy. The original work applied blinking only to a more general distributed memory cache for small objects (Memcached), and thus had no need for replicating objects based on their popularity. We also implement a randomized chunk layout and the Least Recently Used (LRU) cache eviction policy to compare with the proposed load-proportional layout and popularity-based eviction policy, respectively.

Hardware Prototype. We construct a small-scale hardware prototype that uses intermittent power to experiment with GreenCache in a realistic setting. Our current prototype builds off our prototype from Blink [93]. However, unlike that prototype, which uses OLPC nodes, our GreenCache prototype uses more powerful, but energy-efficient, Mac minis. We use a small cluster of ten Mac minis running Linux kernel 2.6.38 with 2.4 GHz...
Intel Core 2 Duo processors and 2GB of RAM connected together using an energy-efficient switch (Netgear GS116) that consumes 15W. Each Mac mini uses a flash-based SSD with a 40GB capacity. We use one Mac mini to run the proxy and power manager, whereas we run a cache server and power client on other Mac minis. The proxy connects to a WiMAX base station (NEC Rel.1 802.16eBS) through the switch. We use a Linux laptop with a Teletonika USB WiMAX modem to run as a client. We also use a separate server to emulate multiple WiMAX clients. Our emulator limits the wireless bandwidth, in the same way as observed by the WiMAX card, and plays the YouTube trace described below. The WiMAX base station is operational and located on the roof of a tall building on the UMass campus. However, the station is currently dedicated for research purposes and is not open to the general public.

Similar to [93], we use ACPI’s S3 suspend-to-ram state as the inactive state. We boot each Mac mini in text mode and unload all unnecessary drivers in order to minimize the time it takes to transition into S3. With the optimizations, the time to transition to and from ACPI’s S3 state on the Mac mini is one second. Note that much faster sleep transition times, as low as a few milliseconds, are possible, and would further improve GreenCache’s performance. We select a blink interval of 60 secs, resulting in a transition overhead of $1/60 = 1.66\%$ every blink interval. With the optimizations above, the power consumption of the Mac mini in S3 and S0 is 1W and 25W respectively. Since GreenCache requires one node, running the proxy and power manager, the switch, and WiMAX base station to be active all the time, its minimum power consumption is 46 W, or 17% of its maximum power consumption.

We power the cluster from a battery that connects to four ExTech 382280 programmable power supplies, each capable of producing 80W, that replay the variable power traces described below. To prevent the batteries from over and under-charging we connect the energy source to the battery using a TriStar T-60 charge controller. We also use hardware sensors to measure the current flowing in and out of the battery and the battery voltage. Our experiments use the battery as a short-term buffer of five minutes.

**Client Emulator.** To experiment with a wide range of video traffic, we wrote a mobile client emulator in Java, which replays YouTube traces. For each video request in the trace file, the emulator creates a new thread at the specified time to play the video as per the specified duration. In addition, the emulator also generates synthetic video requests based on various configurable settings, such as available bandwidth, popularity distribution of videos, e.g., a Zipf parameter, viewing length distribution, and recommended list distribution.

**Power Signal.** We program our power supplies to replay solar and wind traces from our field deployment of solar panels and wind turbines. We also experiment with both multiple steady and oscillating power levels as a percentage, where 0% oscillation holds power steady throughout the experiment and N% oscillation varies power between $(45 + 0.45N )\%$ and $(45 – 0.45N )\%$ every five minutes. We combine traces from our solar/wind deployment, and set a minimum power level equal to the power necessary to operate GreenCache’s always-active components (46W). We compress our renewable power signal to execute three days in three hours, and scale the average power to 50% of the cluster’s maximum power.
### Table 7.1. Standard deviation, 90th percentile, and average buffering time.

<table>
<thead>
<tr>
<th>Blink interval (sec)</th>
<th>Std Dev</th>
<th>90th per</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>7.88</td>
<td>21.25</td>
<td>10.39</td>
</tr>
<tr>
<td>60</td>
<td>14.59</td>
<td>41.25</td>
<td>20.58</td>
</tr>
<tr>
<td>90</td>
<td>24.79</td>
<td>66.25</td>
<td>29.44</td>
</tr>
<tr>
<td>120</td>
<td>30.52</td>
<td>78.25</td>
<td>32.73</td>
</tr>
</tbody>
</table>

### 7.6 Experimental Evaluation

We first benchmark GreenCache’s proxy and chunking overhead for our prototype. We then evaluate GreenCache’s performance for real-world YouTube traces at multiple power levels with varying levels of oscillation. We then demonstrate the performance using realistic power traces from our energy harvesting deployment that have varying power and oscillation levels.

We use two metrics to measure the performance: (1) bandwidth usage between the cache and YouTube servers and (2) average buffering or pause time at the clients. Bandwidth usage denotes the total data received from backend servers over a given time interval; it also represents bandwidth cost that mobile operators must pay to Internet service providers. One primary objective of GreenCache is to reduce this bandwidth usage. Another key objective of GreenCache is to improve user’s viewing experiences. Therefore, we consider average buffering time per video as our second metric to measure the performance. Note that our implementation tries to optimize both metrics independent of each other. However, note that optimizing for bandwidth usage does not depend on the power level, but on the total cache size, while optimizing for buffering time depends on both the cache size and the power level.

#### 7.6.1 Benchmarks

To measure the proxy’s overhead, our client emulator creates a single thread and sends multiple video requests in succession. The breakdown of the latency overhead at each component for a sample 1 MB video chunk of 1 minute play length, assuming a 135 Kbps bit rate, is 30 ms at the proxy, 20 ms at the cache server, 50 ms in the network between the proxy and cache server, and 100 ms in the network between the proxy and client. The result demonstrates that the proxy’s latency overhead is low. We also benchmark average buffering time for different blink intervals at various power levels. Table 7.1 shows the standard deviation, 90th percentile, and average buffering time for video requests, as the
blink interval and power levels change. As expected, the buffering time increases with the blink interval at low to moderate power levels. We also benchmark the standard deviation, 90th percentile, and average buffering time for requests going to YouTube servers, which are as 150ms, 570ms, and 620ms, respectively.

To study the performance of our prototype cache for different cache sizes and power levels we take a 3 hour trace (from 7 PM to 10 PM on February 7th, 2012) from our 3 day YouTube trace. The trace contains a total of 8815 requests, for 6952 unique videos, over the 3 hour interval. Our trace reports the URL, video ID, client IP address, and request time for each video. In addition, we pull the recommended list for each video in the trace from the YouTube servers. Based on the video ID, its recommended list, client IP address, and the next requested video ID, we calculate the viewing length for each video. We assume the average video length as 5 minutes and the streaming rate as 135 Kbps. Also, we fix the downlink bandwidth from backend YouTube servers to the WiMAX station to 1 Mbps, and the storage capacity of each cache server as 1 GB. Further, we fix the blink interval as 60 seconds. We use a weighing factor of 0.6 for the proposed popularity-aware eviction policy.

First, we study the performance—bandwidth usage and buffering or pause time for clients—for different number of cache servers at full power for the real world 3 hour YouTube trace, as well as a synthetic trace of 8815 requests where each request is for a randomly chosen video from the aforementioned 6952 unique videos. In addition, we
choose least-recently-used (LRU) cache eviction policy for this experiment; further, videos are not chunked. Figure 7.9 plots the total bandwidth usage and average buffering time for both random and real traces. We also plot the optimal performance for real traces assuming we know all requests in advance. The optimal policy always keeps most popular videos in the cache, and never evicts a popular video to store a less popular video (over a given interval). As expected, the total bandwidth usage and average buffering time over the 3 hour interval decreases as the size or number of servers increases.

Next, to study the benefits of video chunking we measure the performance of three different cache eviction policies—LRU, popularity-aware, and optimal—for the 3 hour real trace at full power and 9 cache servers. Figure 7.10 shows that the performance of Green-Cache’s popularity-aware eviction policy is better (≈ 7%) than that of LRU. Further, video chunking improves (≈ 15%) the performance of all policies as it avoids storing unpopular chunks of popular videos. In all cases, LRU performs worse than others, which motivates our use of a popularity-aware cache eviction policy and video chunking for all further experiments.

7.6.2 Staggered load-proportional blinking

As discussed earlier, the total bandwidth usage over a fixed interval, as long as a request does not go to backend servers for an already cached video, does not depend on the available power level or blinking and layout policies; it only depends on the cache
size and eviction policies. However, buffering time and users’ experiences do depend on the available power, blinking and layout policies. In this section, we study the effects of the power level on the average buffering time, and various optimizations designed to reduce the buffering time. We use the same 3 hour real YouTube trace, as discussed above, and 9 cache servers for all further experiments. Further, we use video chunking and the popularity-aware eviction policy for all experiments.

To compare the proposed staggered load-proportional policy with the activation and load-proportional policies from Blink [93], we also implement an activation policy and a load-proportional policy for GreenCache, and integrate them with GreenCache’s popularity-aware eviction policy, video chunking, and popularity-aware migration policy. The activation policy activates or deactivates servers as power varies, whereas the load-proportional policy distributes the power to servers in proportion to their popularity. Similar to the load-proportional policy, the activation policy also migrates popular chunks to active servers while deactivating servers due to the drop in the power level. Unlike the proposed staggered load-proportional policy, the load-proportional policy from Blink [93] does not replicate video chunks because it does not benefit from replication as it activates all servers at the same time every blink interval.

Figure 7.11(a) shows the average buffering time at different steady power levels. As expected, the activation policy performs better than the load-proportional policy at low power levels since, unlike the load-proportional policy, the activation policy does not incur the blinking overhead, which becomes significant in comparison to the active interval at low power levels. However, at moderate to high steady power levels, the benefit of a larger cache size, albeit blinking, dominates the blinking overhead for real-world traces. Furthermore, the buffering time decreases significantly if first chunks are replicated on all servers. Even at low power levels, replication of initial chunks significantly reduces the buffering time, while still leveraging the benefits of a larger cache size. Moreover, the performance of the staggered load-proportional policy remains almost the same at all power levels. As video popularity changes infrequently, migration overheads in our experiments are modest (~2%).

Figure 7.11(b) compares the average buffering time for the above policies at different oscillating power levels. We oscillate available power every five minutes. Since migration overhead of the staggered proportional policy is independent of power level, its performance remains almost the same at all oscillation levels. However, the activation policy incurs migration overhead whenever the number of active servers decreases. Consequently, the activation policy performs poorly at high oscillation levels, as indicated in the figure. Though replication of initial chunks reduces the buffering time at all power levels, it is primarily required at low power levels.

Next, we evaluate the benefits of prefetching initial chunks of related videos. As Figure 7.12 indicates, prefetching initial chunks of the top five videos reduces the buffering time by 10% as compared to no prefetching. Further, since prefetching more videos doesn’t improve the buffering time, we limit the cache to prefetching only first chunks of top few videos from the related list. We choose to prefetch top five videos only in order to strike a balance between the performance gain and prefetching overhead.
Figure 7.13. Buffering time at various power levels for our combined solar/wind power trace.

7.6.3 Case study

To experiment with our WiMAX base station using a real WiMAX client, we use a Linux Desktop with Intel Atom CPU N270 processor and 1 GB RAM connected to Teltonika USB WiMAX Modem. We disable all network interfaces except the WiMAX interface. The desktop connects to the WiMAX base station (NEC Rel.1 802.16eBS), which we configure to route all video requests from the desktop to the proxy. We replay the same 3 hour YouTube trace on the WiMAX client, but we use real power traces from our solar/wind deployment, as described in the previous section, to power the GreenCache cluster.

Figure 7.13 plots average buffering time, calculated every five minutes, for three blinking policies: activation, load proportional, and staggered load proportional with first chunks replicated. As expected, the performance of all three policies goes down (buffering time goes up) when the available power drops down, and vice versa. However, the performance of activation degrades more than that of load-proportional when the available power drops down, since the activation policy incurs migration overhead when the number of active servers decreases. Further, replicating first chunks significantly reduces the buffering time for the staggered load-proportional policy at all power levels. Since the migration overhead of the staggered load-proportional policy is independent of power levels, its performance does not vary much, not even when the available power changes significantly, if first chunks are replicated.

7.7 Related Work

The use of caches to improve the performance of multimedia distribution systems has been studied extensively in the past two decades. [98] gives a general overview on existing multimedia caching techniques. Due to the vast amount of exiting work in this area, we only focus on the work closely related to our approach, although, to the best of our knowledge, there is no existing work that directly addresses multimedia caches for intermittent power.

Wu et al. [101] were among the first to propose the caching of chunks (segments) of a video. In contrast to our approach chunks are not equal in size and increase exponentially with the distance from the start of the video. The intention of this approach is to combine
the number of consecutive chunks that are cached with the popularity of the video. E.g., for a very popular video all chunks would be stored on the cache while for less popular chunks only a certain number of the initial chunks of the video would be cached. Letting the chunk size grow exponentially has the advantage that the initial chunks of many videos can be stored without occupying too much of the caches storage space. Having only one or several initial chunks of a video stored on the cache bears the advantage that a requested video can be streamed to the client and played out without significant delay. Missing chunks can be streamed from the server immediately after the initial client request to allow for a smooth play out. In contrast to the approach presented by Wu et al., we decided for a scheme that splits all videos in equal sized chunks (except for the very last chunk) where the complete chunk can be transmitted to the client in a period that is equal or smaller than the blink interval, assuming a minimum transmission rate.

A more restrictive version of the caching of video chunks is the caching of the first chunk (prefix) only, which was introduced by Sen et al. [92]. The sole goal of this approach is to reduce the buffer time at the client, since the first chunk can be streamed from the cache much faster than from a remote server. As shown in [90], prefix prefetching can significantly improve the viewer’s experience of watching videos and this motivated us to investigate how the prefetching approach performs on a multimedia cache for intermittent power. The results presented above show that prefix prefetching can improve the experience of a viewer also in the case of a blinking multimedia cache.

As in our current work, trace-based driven simulations are also used in [87] and [103] to investigate the effectiveness of caching for YouTube videos. Both investigations show that caching of YouTube video can both, on a global and regional level, reduce server and network load significantly. In contrast to the work presented in this chapter, both studies do not consider scenarios in which power for the caches is intermittent.

7.8 Conclusion

This chapter presents techniques for optimizing multimedia caches running off intermittent renewable energy sources. These caches are important in improving the performance of “off the grid” cellular towers and base stations, which are increasingly common in developing countries that have both a lack of infrastructure and an unstable grid. We show how combining a blinking abstraction, which rapidly transitions cache servers between an active and inactive state, with techniques for chunking (or segmenting) videos, replicating popular chunks, and staggering server active intervals improves the performance of these caches when compared to standard techniques for powering servers on and off based on available power. Our work demonstrates that running multimedia caches off intermittent power from renewables poses interesting new research problems. Our results show that GreenCache’s blinking techniques decrease both backhaul bandwidth and client access latency compared to existing approaches that simply activate and deactivate servers. In the latter case, resulting in 3X less buffering (or pause) time by the client watching a video.
CHAPTER 8
SUMMARY AND FUTURE WORK

8.1 Thesis Summary

This thesis has explored how distributed systems can be redesigned to run solely on renewable sources of energy while performing well in spite of significant and frequent variations in power supply. Further, I have also explored how weather forecasts, which provide better predictions than PPF at short to medium time scales ranging from 1 hour to 3 days, could be leveraged to predict future solar and wind energy harvesting.

First I analyzed historical weather forecast data from NWS and found that weather forecasts are a better indicator of future weather than the immediate past. Unlike PPF techniques, weather forecasts can also predict sudden changes in weather conditions, which happen frequently in many parts of the country including our site. I have designed two simple empirical models to predict solar and wind energy harvesting based on sky cover and wind speed forecasts from NWS, and shown that predictions based on weather forecasts outperform predictions based on immediate past by more than 20% in all weather conditions.

Next I showed how machine learning techniques could be used to better understand relationships between different weather parameters and how they impact solar intensity. I found that sky cover is not the only parameter that affects solar intensity; rather other weather parameters, such as temperature, precipitation potential, etc., affect solar intensity too. I used the statistical power of machine learning techniques to design better prediction models for solar energy harvesting. Unlike previous empirical models, machine learning models do not require solar panel and wind turbine deployment to derive region specific models. Further, I found that SVM-based prediction models are 27% more accurate for our site than previous models for solar energy harvesting.

To manage server clusters on renewable sources of energy I proposed an energy abstraction, called Blink, to regulate the energy footprint of servers in response to variations in power supply. I also designed various blinking policies to coordinate blinking across servers in a server cluster. Though each blinking policy allows instant capping of the cluster-wide power consumption to the available power, different applications can use different policies to optimize their performance for different workloads. To see how a real-world application can leverage the blinking abstraction I modified Memcached – an in-memory key-value storage system for small chunks of arbitrary data. I showed that a real-world application could be redesigned, with modest complexity, to perform well on intermittent power.
I assume Blink provides a general abstraction for regulating energy footprint of servers, which can be used by a range of distributed applications to experiment with intermittent power constraints. To extend the horizon of the application-agnostic blinking abstraction across a wide range of real-world applications, I designed and implemented BlinkFS, a blink-aware distributed file system that combines blinking with a power-balanced data layout and popularity-based replication/reclamation to optimize I/O throughput and latency as power varies. I experimented with three distributed applications – Memcache, MapReduce, Search Engine – and found that unmodified real-world applications can perform well in spite of significant and frequent variations in the available power. Moreover, BlinkFS outperforms existing distributed file systems for both a sustained low power scenario as well as a highly intermittent power scenario.

Finally, I used the blinking abstraction to design a distributed multimedia cache, called GreenCache, for intermittent power. I designed several optimization techniques to minimize backhaul bandwidth usage while reducing buffering time at all power levels. I also analyzed extensive traces of YouTube videos and used users’ viewing characteristics to design popularity-aware cache eviction and prefetching policies. I found that GreenCache’s staggered load-proportional blinking policy in combination with a popularity-aware cache eviction policy results in 3X less buffering (or pause) time by the client compared to an activation blinking policy, which simply activates and deactivates servers over long periods as power fluctuates, for realistic power variations from renewable energy sources.

Our results show that a number of real-world applications can leverage the blinking abstraction to handle intermittent power constraints. Further, many distributed systems can be redesigned, with modest complexity, to perform well on intermittent power, and thus can reduce their dependency on dirty fossil fuels and increase renewable energy penetration without sacrificing the performance.

8.2 Future Work

This section introduces some future research directions that can extend the work presented in this thesis.

- **Machine Learning Model For Wind Energy Harvesting:** As I have discussed in this thesis, machine learning provides a few powerful techniques to analyze the correlation between solar intensity and other weather parameters, and use the forecasts of other parameters to predict solar energy harvesting in the future. We can explore these techniques to study the correlation between wind speed and other weather parameters, and design a similar prediction model for wind energy forecast based on forecasts of other weather parameters. Further, these techniques could be expanded well beyond predicting solar and wind energy harvesting. For example, similar techniques could be devised to predict spot market prices of electricity in wholesale electricity markets.

- **Leveraging Energy Harvesting in Blink-Aware Distributed Systems:** In this thesis, I have designed distributed systems to run on intermittent power, assuming the
available power can change unpredictably and we have no knowledge of future energy supply. One can use energy forecast from our energy harvesting prediction models and use that knowledge to schedule applications and data migration to further improve the performance of applications running on a blinking cluster. In addition, energy harvesting knowledge could also be used to make energy-purchasing decisions in hybrid data centers, running on both renewables and grid energy, that will optimize the energy budget while satisfying SLAs.

- **Leveraging Blinking Abstraction in Grid-Powered Data Centers:** Another compelling use of intermittent power is continuously operating a data center power delivery infrastructure at 100%. Since data center capital costs are enormous, maximizing the power delivery infrastructures utilization by operating as many servers as possible is important. However, data centers typically provision power for peak demands, resulting in low utilization. In this case, intermittent power is useful to continuously run a background workload on a set of servers – designed explicitly for intermittent power – that always consume the excess power PDUs are capable of delivering. Since the utilization (and power usage) of a data centers foreground workload may vary rapidly, the background servers must be capable of quickly varying power usage to not exceed the power delivery infrastructures limits. Deciding which applications to run on blinking nodes and when to run in order to maximize the overall performance of the data center is a challenging problem and requires significant research contribution.
BIBLIOGRAPHY


