Beyond Energy-Efficiency: Evaluating Green Datacenter Applications for Energy-Agility

Supreeth Subramanya, Zain Mustafa, David Irwin, and Prashant Shenoy
University of Massachusetts Amherst

ABSTRACT

Computing researchers have long focused on improving energy-efficiency under the implicit assumption that all energy is created equal. Yet, this assumption is actually incorrect: energy's cost and carbon footprint vary substantially over time. As a result, consuming energy inefficiently when it is cheap and clean may sometimes be preferable to consuming it efficiently when it is expensive and dirty. Green datacenters adapt their energy usage to optimize for such variations, as reflected in changing electricity prices or renewable energy output. Thus, we introduce *energy-agility* as a new metric to evaluate green datacenter applications.

To illustrate fundamental tradeoffs in energy-agile design, we develop GreenSort, a distributed sorting system optimized for energy-agility. GreenSort is representative of the long-running, massively-parallel, data-intensive tasks that are common in datacenters and amenable to delays from power variations. Our results demonstrate the importance of energy-agile design when considering the benefits of using variable power. For example, we show that GreenSort requires 31% more time and energy to complete when power varies based on real-time electricity prices versus when it is constant. Thus, in this case, real-time prices should be at least 31% lower than fixed prices to warrant using them.

1. INTRODUCTION

Energy-efficiency, which is defined as the amount of work, i.e., computation and I/O, done per joule of energy, has long been considered a "first class" metric for evaluating computer system performance. Energy-efficiency has become particularly important for warehouse-scale datacenter facilities, since a greater energy-efficiency reduces these facilities' large electric bills (assuming that utilities charge a constant price for energy over time) and their carbon emissions (assuming all their energy is created from carbon-based sources). Energy-efficiency for warehouse-scale datacenter facilities remains a highly active research area, as their size and number continues to grow to satisfy the demand for cloud-based services. The power requirements of the largest datacenters now exceed 100 megawatts (MW) [12], and, collectively, they are estimated to consume 1.7-2.2% of U.S. electricity [20].

To reduce energy's carbon footprint and cost, green datacenters are experimenting with generating clean power locally from renewables [5, 13] and participating in utility demand response (DR) programs [24, 40], which offer reduced rates if consumers respond to signals (often in the form of higher electricity prices) to curtail their energy usage. Both renewable energy and DR introduce the potential for variations in available power. As others have noted [19, 40], data centers are well-equipped to respond to such variations, since they i) already include sophisticated power management functions, which are remotely programmable and capable of varying power usage over a wide dynamic range, and ii) often execute non-interactive batch applications that are tolerant to some delays due to power shortages.

Unfortunately, as a metric, energy-efficiency does not account for variations in energy's cost and carbon footprint, but rather implicitly assumes i) that all energy is created equal and ii) that it is available in unlimited quantities at any time. These assumptions are not correct: in reality, all energy is not created equal—its cost and carbon footprint vary over time depending on the mix of generators used to to create it—and, as the reliance on intermittent renewable energy increases, it may not be available in unlimited quantities. Thus, just because a system is highly energy-efficient does not necessarily mean that its cost and carbon footprint is lower than a highly inefficient one. That is, an inefficient system that consumes energy at the "right" times, e.g., when renewable energy is plentiful or electricity prices are low, could be cleaner and cheaper than an energy-efficient system that uses energy at the "wrong" times, e.g., when renewable energy is scarce or electricity prices are high.

Thus, energy-efficiency is not the right metric to quantify the performance of green datacenter applications that adapt to a variable supply of power. To properly evaluate these applications, we propose a new metric, which we call *energy-agility*. *While energy-efficiency is a measure of work done per joule of energy consumed by a platform, energy-agility is a measure of work done per joule of energy available to a platform, which may vary dynamically over time.*

Thus, as a metric, energy-agility captures the salient characteristics above that i) energy is not always available in unlimited quantities at any time, and ii) the availability of energy may vary over time. Note that the energy available to a platform is independent of how much energy it actually consumes. Whereas energy-efficiency only depends on

ICPE'16, March 12 - 18, 2016, Delft, Netherlands
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4080-9/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851553.2851556

how much energy a platform consumes to perform a given amount of work, energy-agility applies a "use it or lose it" property to energy that incentivizes platforms to use as much energy as possible, as efficiently as possible, when it is available, or else waste it. Thus, energy-agility depends on how much energy is available to a platform to perform a given amount of work, regardless of the amount of energy that it is able to productively consume. Energy-agility captures the basic characteristic that electricity's supply and demand must be balanced at all times, and the only way to not waste unused energy is to explicitly store it for later use.

To illustrate fundamental tradeoffs in energy-agile design, we develop GreenSort, a distributed sorting system optimized for energy-agility. GreenSort is representative of the long-running, massively-parallel, data-intensive tasks that are common in datacenters and amenable to delays from power variations. Unlike short batch jobs, which a scheduler may simply defer until power is plentiful or cheap enough to complete them [16, 17], such "big data" applications must adapt their execution in real time by continuously modifying their energy usage to not exceed the available supply. In developing GreenSort, we make the following contributions.

Energy-Agility Metric. We introduce energy-agility as a new performance metric that is distinct from energy-efficiency, and motivate its importance in evaluating emerging green datacenter applications that use variable power.

GreenSort Design. We design multiple GreenSort variants to illustrate fundamental tradeoffs in energy-agile design of a prototypical datacenter application. Each variant is defined by a power management policy that performs well for a particular area of the design space, which is defined by the power signal characteristics, power state transition latencies, energy storage capacities, input data size, etc.

Implementation and Evaluation. We implement and evaluate GreenSort to quantify its performance. We demonstrate the extent to which power variations increase the time and energy to complete a task, which highlights the importance of energy-agile design when considering the benefits of using local renewables or participating in DR programs. For example, we show that GreenSort requires 31% more time and energy to complete when power varies based on real-time electricity prices versus when it is constant. Thus, in this case, real-time prices should be at least 31% lower on average than fixed prices to warrant opting into using them.

2. BACKGROUND

There has been a variety of recent research on designing green datacenter applications that adapt to power variations due to changing electricity prices or renewable energy output. For example, prior research has focused on optimizing a variety of system components for variable power, including distributed caches [32], file systems [33], virtual machines [23, 34], and batch schedulers [3, 15, 16, 17, 21]. Prior research has also investigated the use of energy storage to dampen or eliminate the effect of power variations [18, 39].

Since energy-efficiency alone does not capture the benefits of using variable power, the metric these systems measure themselves against is generally the cost of energy, as variable electricity prices are typically lower, on average, than flat prices. Thus, the "performance" benefit of prior systems is largely dependent on the absolute price of electricity: the more variable the prices, or the wider their range, the more cost savings are possible. However, energy's cost is not a sound basis for evaluation, since energy prices vary significantly by region, by time, and based on external factors. Rather, cost is only useful for assessing the monetary benefits of employing a particular system or approach at a specific point in time. Energy-agility provides a metric independent of cost to evaluate and compare the performance of such systems, similar to how the absolute cost of energy has (and should have) no bearing on a system's energy-efficiency.

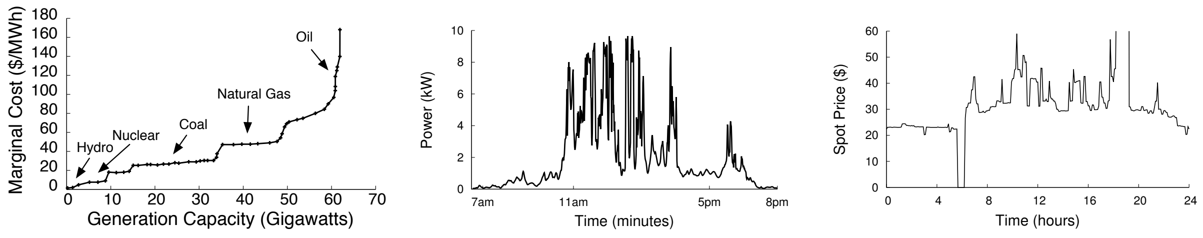
2.1 Energy-agility Definition

Formally, energy-agility is a measure of the amount of work W, e.g., computation and I/O, done by a system given a *power signal* $P(t)$ that dictates an energy cap the system must adhere to over each interval $(t-\tau, t]$ for some interval length τ. Energy-agility does not dictate the underlying reason for the power variations, e.g., due to DR signals, fluctuations in renewable generation, changes in electricity prices, etc., or the characteristics of $P(t)$, which may differ widely depending on the scenario. We discuss different types of emerging scenarios and power signals in §2.2. As we show, $P(t)$'s characteristics—its average magnitude, variance, and range—have a significant influence on energy-agility design. Also, note that energy-agility incorporates energy-efficiency, and is not entirely orthogonal to it: to maximize energy-agility, at any given time t, an application is always incentivized to use the available energy as efficiently as possible.

For sorting, energy-agility translates to the number of records sorted per joule of energy *made available to* a platform over its running time, whereas energy-efficiency is the number of records sorted per joule of energy *consumed by* a platform over its running time. Thus, to complete a given amount of work, a greater energy-agility translates into a shorter running time and the need for less aggregate energy to be made available. The value τ derives from the minimum energy storage capacity necessary to enforce a platform's maximum energy cap over each τ. We assume $\tau > 0$, since platforms require some minimal energy storage capacity, as they cannot respond instantaneously to changes in power due to inherent latencies in transitioning power states. Due to energy's "use it or lose it" property, our definition dictates that application's waste any energy they cannot immediately use or store. Since storing energy incurs inverter and conversion losses, we assume an application loses a fraction L of any energy stored beyond the next interval τ. A typical value of L for lithium-ion or lead-acid batteries is 0.2.

The primary intent of energy-agility as a metric is to enable systems designers to reason about the effect of power variations on application performance. Intuitively, a more stochastic $P(t)$ increases an application's running time and, thus, the aggregate energy it requires to perform a given amount of work. As a result, even if variable energy is cheaper per kilowatt-hour (kWh), an application may not be cheaper overall to execute with variable power if it must run longer and consume more energy to complete.

Of course, the more energy storage capacity available, the more an application can dampen any power variations. Unfortunately, energy storage is highly expensive. Thus, an application that achieves the same performance using little or no energy storage, e.g., by adapting its power usage in real time, is preferable to one that uses significant energy storage. Ideally, systems would perform the same regardless of the characteristics of $P(t)$; that is, they would perform the same amount of work for a given amount of available



(a) Generator Operating Costs (b) Example Solar Output (c) Electricity Prices

Figure 1: The use of heterogeneous generators with different marginal costs (a) and the rising penetration of intermittent renewable energy sources (b) cause fluctuations in electricity's price every few minutes (c).

energy over time, regardless of how it varies. In practice, however, systems are not ideal: they require non-trivial latencies to toggle between numerous power states to cap their energy usage, which incurs overhead and affects application performance. Thus, quantifying the performance overhead caused by adapting to variable power is important in assessing the benefits of both energy storage and using variable power relative to its cost.

2.2 Emerging Scenarios

The motivation for energy-agility ultimately derives from the fact that all energy is *not* created equal [36]: instead, it derives from a heterogeneous mix of generators with different fuel costs, carbon emissions, and operational characteristics. For example, while solar and wind farms have variable energy output over time, they have no associated fuel costs or carbon emissions. Thus, consuming energy when renewables are generating it results in less carbon emissions than consuming it when they are not generating it. As another example, the "peaking" generators that utilities dispatch to satisfy transient demand peaks have much higher emissions and fuel costs (>10x [11]) than the baseload generators they continuously operate. Figure 1 illustrates the point by showing the disparity in marginal cost of operating various types of generators and the rapid fluctuations possible with renewable energy, which both contribute to large energy price fluctuations over time.[1] Such fluctuations are expected to intensify as renewable penetration increases in the grid.

Prior work on energy-efficient computing largely ignores how energy is actually generated in the electric grid. This is due, in part, because i) renewable energy is only now becoming a viable alternative to carbon-based energy sources and ii) utilities have historically masked variations in energy's cost and carbon footprint from consumers by charging them a fixed price for energy over time. However, there are now numerous examples of data centers using energy from local or nearby large-scale solar and wind farms [5, 13] with the most prominent example being Apple's new iCloud data center, which includes a 20MW co-located solar farm [5]. These facilities may need to vary their power usage based on renewable generation if they cannot make up the difference from other sources, e.g., energy storage or the grid.

In addition, with the mass deployment of smart electric meters, which record and transmit electricity usage in real-time at fine-granularities, e.g., every 15 minutes or less, utilities are beginning to implement more sophisticated pricing and DR mechanisms. For example, many consumers may now opt into real-time pricing, where electricity prices change every few minutes based on supply and demand. For large industrial energy consumers like datacenters, there are other DR programs available. For example, datacenters might act as *load resources* (LRs) in the grid's ancillary service markets [40]. Somewhat like a "reverse generator," the grid controls LRs to modulate their energy "generation" by signaling them to increase and decrease power usage over time. LRs receive compensation based on their ramp time (the time required to effect a change in power) and capacity (the range of power over which they have control). Datacenters are well-suited to act as LRs, since they have short ramp times and high capacities. The grid will require more LR capacity, as renewable penetration rises, to balance an, increasingly stochastic, supply with demand.

Each of the scenarios above introduce *hard power caps* dictated by $P(t)$ with potentially different characteristics, e.g., solar power is likely more periodic and less stochastic than wind power. In contrast, prior work that is cost-oriented often assumes only soft power caps, such that, while using power may be undesirable because it is expensive or "dirty," it is always available if necessary at some price [16, 17, 24].

2.3 Power Capping Mechanisms

A prerequisite for capping energy over each τ is a mechanism to cap server power. A variety of active [7, 14] and inactive [32, 37] power capping mechanisms exist, although the specific mechanisms available are platform-dependent.

Active power capping bounds power usage by reducing servers' performance without deactivating them; it primarily focuses on reducing CPU power usage using a combination of dynamic voltage and frequency scaling (DVFS) and transitioning CPUs into low-power idle modes, e.g., ACPI's C-states. Recent research applies similar concepts to actively cap memory power [9]. While active power capping incurs low overhead, since transitions between active power states are rapid, e.g., milliseconds or less, it generally is able to lower server power usage to at most 50% of peak power [4, 37]. Unfortunately, active power capping does not reduce the power usage of other power-hungry components, such as the motherboard, disk, network card, etc. The narrow power range offered by active power capping is one reason reducing the power usage of interactive applications with low latency requirements has proven challenging [25, 27].

Inactive power capping bounds power usage by transitioning servers to an inactive power state, which deactivates a server by cutting power to nearly every component and reducing server power to near zero. For example, ACPI's Suspend-to-RAM (S3) state preserves DRAM memory state, but turns off all other components, while its Suspend-to-

[1]Data for Figure 1(a) is from [11]; (b) is from a 10kW home solar installation; and (c) is from a representative week in the New England five-minute spot market.

Disk (S4) state writes memory state to disk before turning off the server. Prior work argues that inactive power capping is more efficient than using active power capping, assuming low (<100ms) transition latencies [26]. Unfortunately, while the precise time to transition to and from an inactive power state is platform- and OS-dependent, it typically takes between tens of seconds (for S3) to a few minutes (for S4) [1, 32]. A server cluster may implement inactive power capping by either transitioning some subset of servers to the inactive state based on the available power [37], or by "blinking" servers between active and inactive states in tandem to cap power over short intervals [32]. With the latter approach, servers are inactive for some fraction of each interval based on the average power available over the interval.

In this paper, we assume that both active power capping and inactive power capping, either via deactivating or blinking servers, are available, although the precise power range and overhead of each mechanism varies widely by platform.

3. GREENSORT DESIGN

To illustrate tradeoffs in designing energy-agile applications, we develop GreenSort, a distributed sorting system optimized for energy-agility. The primary constraint GreenSort adds to prior sorting applications is the power signal $P(t)$ that dictates an energy cap the sorting platform must adhere to over each interval $(t-\tau, t]$. We choose distributed sorting to illustrate energy-agile design for a variety of reasons. Most importantly, sorting lies at the core of many "big data" frameworks, including MapReduce [8]. Sorting is also a particularly demanding workload, as it requires shuffling its entire input dataset across all servers. A similar all-to-all shuffling phase is the bottleneck for many data-intensive applications. Finally, sorting stresses a mix of system resources: it is largely I/O-bound, but also requires non-trivial CPU time and, if distributed, network bandwidth.

GreenSort follows in a long line of sorting systems that highlight various aspects of systems design, such as I/O performance, energy-efficiency, and cost, to motivate researchers to improve upon them [35]. Insights from these prior sorting systems have influenced a broad range of systems. For instance, the notion of balance in JouleSort [31] influenced the design of energy-efficient key-value stores [4], MapReduce platforms [29], and databases [38].

Since, from an algorithmic standpoint, sorting is a solved problem with well-established tight performance bounds [2], results from prior sorting systems [31, 30, 35] primarily represent a measure of a hardware platform's capabilities combined with various software optimizations that best exploit those capabilities. GreenSort differs from prior systems in this respect: while it also represents a measure of hardware capabilities, particularly the set of available platform power states and the time to transition between them, it alters the sorting problem due to the use of inactive power states. Thus, in addition to minimizing I/O, as in conventional external sorting, GreenSort must also consider the effect of transitioning power states, which are time-consuming and may periodically render some data unavailable.

To understand how designing for energy-agility affects distributed sorting, we first summarize the design of conventional distributed sorting on an always-on cluster with no power constraints (§3.1). We then present multiple GreenSort design variants that optimize the conventional design for energy-agility under different conditions, as defined by the power signal, power state transition latencies, energy storage capacities, input data size, etc (§3.2).

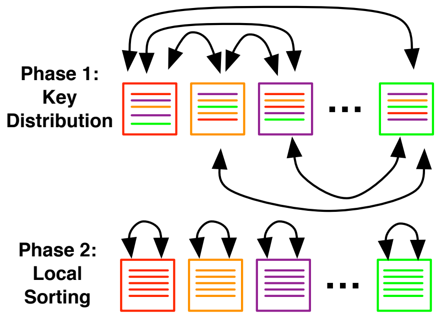


Figure 2: A conventional distributed sort has a key distribution phase followed by a local sorting phase.

3.1 Conventional Distributed Sorting

We assume a cluster with N homogeneous servers, each with some local storage to hold the unsorted input and the sorted output. We also assume the input data is significantly larger than the collective memory of the N servers, requiring the servers to store the input and output on disk. Thus, sort performance is largely dictated by storage I/O bandwidth, where an efficient distributed sorting implementation generally uses a large number of local disks in parallel, typically more than one per server. Initially, each server stores a random set of $1/N$th of the input records to be sorted.

Without loss of generality, we assume Indy sorting, where i) each record is 100 bytes where the first ten bytes serve as a random key and ii) keys are initially distributed uniformly across servers, mitigating the need for a separate key sampling phase to determine the distribution. The sort application divides the key space across the N servers, such that, when sorted, keys in the range $[(i-1)/N, i/N)$ are in sorted order on the ith node. The input and output are stored as files on the N servers, with the concatenation of the output files representing the sorted input.

We use the term *worker* to refer to the sorting application process on each server. Given the setup above, a conventional distributed sort may proceed in two phases [30], as depicted in Figure 2: a key distribution phase followed by a local sorting phase. During key distribution, each worker i) sequentially reads the list of keys from disk and sends them to their destination server and ii) receives keys destined for it from other workers and writes them to disk. Workers divide their key space across multiple separate files, such that each file stores a separate sub-range of the keys and fits into the workers' memory. Once each worker finishes key distribution, it locally sorts keys by reading each file into memory, sorting it with a textbook in-memory sorting algorithm, and writing it back to disk. Since our input is much larger than the servers' memory, the amount of disk I/O dictates sorting performance. This conventional design uses the theoretical minimum number of I/Os with two reads and two writes per record [2]: each record is read once at its origin server, written to a file at its destination, read again for in-memory sorting, and finally written again to the final sorted output.

3.2 Energy-Agile Distributed Sorting

As power rises and falls, an energy-agile sort must determine how to best divide the available power among the servers to finish the sort as fast as possible. Thus, in addition to workers, GreenSort also employs a *power manager* that partitions available power every τ across servers using the mechanisms from §2.3. For GreenSort's design, we focus on the key distribution phase, since workers must coordinate with each other to exchange keys during this phase. Thus, degrading the performance of one worker, e.g, via power capping, during key distribution affects the performance of the other workers. In contrast, since the local sorting phase is embarrassingly parallel, the power manager need only ensure that it allocates power to workers with work remaining.

The power manager may be able to satisfy small drops in power using active power capping without affecting the sorting application's operation. When using active power capping, the optimal strategy is to maintain *balanced progress* across all $\binom{N}{2}$ pairs of workers exchanging keys by dividing the available power equally across servers, such that each server has the same power cap. Since all workers must exchange keys with all other workers, if any worker is slower than the others, due to being in a lower power state, it will create a bottleneck by slowing the progress of all workers in distributing keys. However, significant drops in power require the sorting application to use inactive power capping, which renders some servers unavailable. Deactivating servers complicates the key distribution phase, since it disrupts the all-to-all data shuffle among servers. Below, we describe policies for capping power by deactivating servers in priority order and by blinking, as well as their advantages and disadvantages when shuffling keys across servers.

3.2.1 Priority Policy

A straightforward approach to inactive power capping is to activate and deactivate a subset of servers in priority order based on the available power. This policy uses the available power $P(t)$ each interval τ to activate servers such that it minimizes "wasted" power, while enforcing an equal allocation of power across servers to maintain balance.

We consider power as wasted if it is either below a server's minimum active power or above its maximum power necessary to run an uncapped worker. In the former case, the server has only enough power to turn on and can do no useful work, while, in the latter case, the server cannot make use of any power above some maximum value. For example, assume a power cap of 400W for a cluster of five servers, which have a minimum idle power of 50W and a maximum power of 150W when executing an uncapped worker. In this case, activating five servers at 80W wastes the most power (250W), leaving the least for useful work (150W), while activating three servers at 133W wastes the least power (150W), leaving the most for useful work (250W).

After determining how many to activate, the policy must then determine which servers to activate. To do this, the policy prioritizes servers (arbitrarily) from $1 \ldots N$, such that the highest priority server not yet completed with the key distribution phase always remains active, assuming enough power to activate one server. As power increases, the policy activates the next highest priority server(s) not yet finished exchanging keys with the current highest priority server. Likewise, as power decreases, the policy deactivates the lowest priority active servers. Once the highest priority server finishes its key distribution phase by exchanging keys with all other servers, the policy deactivates it and places it at the lowest priority, resulting in a new highest priority server.

Note that the priority policy incurs minimal overhead to transition servers to the inactive power state, since it only transitions a server to an inactive power state if i) power increases or decreases or ii) a server finishes the first phase and has no more work to do. In fact, assuming only constant power to activate k of n servers, the minimum number of transitions is $\sum_{i=0}^{\lceil \frac{n}{k-1} - 1 \rceil} (n - i(k-1)) = O(n^2/k)$. This amount represents a lower bound on the minimum number of transitions if power varies, since increases and decreases in power may force additional transitions. Thus, for variable power, the minimum number of transitions is a function of the power signal's variability. However, as we discuss below, minimizing the transitions comes at a cost: it results in imbalanced progress and requires significant modifications to the conventional distributed sorting implementation.

Imbalanced Progress. Unlike with active power capping, the priority policy results in maximum imbalance in the progress of exchanging keys, since high priority servers finish before low priority servers have begun. While such imbalance is not an issue for constant power, since any work done is useful, it results in wasted work if power is highly variable.

To understand why, consider a simple scenario where there is enough constant power to activate two servers at all times. In this case, the priority policy cycles through each distinct combination of servers, while, each time, fully completing each pairs' key exchange. As a result, at any given time, some set of two-server combinations has completely finished exchanging keys, while the remaining two-server combinations have not yet started. Thus, if it takes $T_{exchange}$ time for two servers to finish exchanging keys *and* if power ever increases to full power (sufficient to power all N servers), then it will still take $T_{exchange}$ time to complete the key exchange for the remaining two-server combinations not yet started. Thus, any work completed at low power levels is effectively wasted: *had no servers exchanged keys until the time of the power increase, the total running time of the key distribution phase ($T_{exchange}$) would be the same.* Since the priority policy is not balanced, the remaining running time of the key distribution phase is always based on the lowest-priority pair of servers with the least progress.

Sorting Modifications. The priority policy also requires modifications to the sorting application, since servers are no longer always concurrently active. As a result, each worker running on an active server must maintain an up-to-date list of the other active workers to ensure that it only attempts to exchange keys with those workers. Of course, since input records are stored in random order on disk, an active worker that is sequentially reading records to distribute to other active workers will invariably read records destined for currently inactive servers. While a worker may be able to briefly cache these records in memory, it will ultimately have to write them to disk if the destination server does not become active in the near future.

Thus, rather than completing the key distribution phase using a single sequential scan of keys per server (with one read per key), the priority policy requires multiple passes over the keys. Even if, on the first scan, the worker stores a pointer to the location of an inactive servers' discarded keys on disk, such that only one additional read per key is required once a server becomes active, these subsequent reads will result in random, rather than sequential, I/O. Since random I/O bandwidth is generally two orders of magnitude slower than sequential I/O bandwidth, this significantly de-

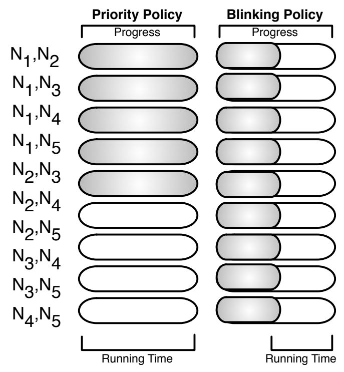


Figure 3: The priority policy causes imbalance during key distribution, while blinking does not.

grades performance of an I/O-bound distributed sort.

We address this problem in the priority policy by pre-partitioning the keys on each server before distributing them, such that the keys for a particular destination server are stored sequentially on disk. This pre-partitioning step incurs more I/O (one additional read and write) to improve performance by eliminating the need to waste I/Os by reading and discarding records destined to inactive servers. However, by sequentially storing keys destined for the same server, it increases the amount of sequential I/O during the key distribution phase.

3.2.2 *Blinking Policy*

The blinking policy differs from the priority policy by dividing each interval τ into an active and inactive period based on the energy available over τ [32]. For example, if τ is two minutes, the average power available over τ is 100W, and each server's active power cap is 200W, then the active and inactive periods would each be one minute. In addition, since blinking concurrently activates all servers each interval τ, it introduces a choice in setting a server's active power cap and the length of the active period each τ. To reduce wasted power, we set the active power cap to the minimum cap possible that maximizes utilization of the CPU (its most energy-efficient setting). Thus, blinking makes minimal use of active power capping, especially on balanced platforms that fully utilize the CPU. Relative to the priority policy, blinking has three main benefits.

Few Application Modifications. Unlike with the priority policy above, blinking requires few changes to the sorting application, itself, although the power manager must compute the active and inactive periods every τ and synchronously toggle servers to and from the inactive power state in tandem. The conventional distributed sort requires no changes, since inactive state transitions preserve memory state and, as before, all servers are always concurrently active

Maintains Balanced Progress. As when using active power capping, blinking servers maintains balanced progress across all workers, such that all workers distribute keys to other workers at the same rate. Since no single worker is ever a bottleneck to finishing the key distribution phase, unlike the priority policy, blinking behaves similarly regardless of the variability of the power signal. Figure 3 depicts this advantage of the blinking policy over the priority policy, where $N = 5$ and power increases to full power at the mid-point of execution. Each bar represents the progress in exchanging keys between each of the $\binom{5}{2}$=10 distinct pairs of servers.

In this case, the figure shows that, on reaching full power, the balanced blinking policy will finish in half the time of the priority policy modulo transition overheads. Notice that imbalanced progress is only an issue if power changes, i.e., drops and then rises again, since the bottleneck only presents itself when power increases. Thus, the priority policy becomes progressively worse as power becomes more variable.

Capable of Low Power Caps. While the priority policy needs at least enough power to activate two servers during key distribution to perform useful work, the blinking policy is able to perform useful work with much less power simply by reducing the length of its active interval.

Unfortunately, blinking also has drawbacks. Most importantly, some non-trivial portion of the active time each interval τ is wasted due to transitioning power states, which may take anywhere between a few seconds to multiple minutes depending on the platform. In addition, since transitions occur at a fixed interval, the number of transitions is based on an application's running time, rather the variability in the power signal. Finally, such frequent transitions may also degrade the reliability of mechanical disks.

3.2.3 *Round-robin Policy*

The blinking and priority policy, represent two extremes in the energy-agility design space captured by Table 1. The blinking policy works well with small input data (resulting a short running time), short transition latencies, highly variable power, and low average power, since it incurs frequent and costly inactive power state transitions but maintains balanced progress between each pair of servers. In contrast, the priority policy works well with larger input data (resulting in longer running times), long transition latencies, less variable power, and a higher average power, since it transitions to inactive power states less but results in imbalanced progress for variable power signals.

We introduce a round-robin priority policy to mind the gap between these two extremes. This policy behaves like the priority policy, in that it assigns priorities to servers $1 \ldots N$ and activates them in order. The primary difference is the round-robin policy defines a scheduling time slice t_{sched}, which sets the maximum time any server may be active. Once a server exhausts its time, the policy deactivates the server, sets it to the lowest priority, and then activates the next highest priority inactive server. With a long time slice, the round-robin policy behaves similarly to the priority policy (with few transitions but imbalanced progress), while with a short time slice, it approximates the blinking policy (with many transitions but balanced progress).

3.2.4 *Performance Modeling*

We model GreenSort's performance based on its salient characteristics: the running time T when using full power with all N available nodes active, the inactive power state transition latency d, the average number of active nodes k based on the power signal $P(t)$, the time for preprocessing data on a given node T_{pre}, and the blink interval τ. M then represents the number of power state transitions required by each policy. Based on these variables, we can define the running time of Greensort under each policy as follows.

$$T * (\frac{N}{k}) + T_{pre} * (\frac{N}{k}) + d * M \qquad (1)$$

Here, the first term represents the sort's running time when only k of N nodes are active on average, since run-

Policy	*Running Time (T)*	*Transition Latency (d)*	*P(t) Variability*	*P(t) Average*
Blinking	Low	Low	High	Low
Round-Robin	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$
Priority	High	High	Low	High

Table 1: Qualitative attributes of a distributed sort that are amenable to the blinking and priority policies. The round-robin policy's configurable time slice minds the gap between the two policies.

ning time is a linear function of the number of active servers. The second term is the time spent preprocessing input data prior to key distribution. The last term is the overhead due to power state transitions over the running time. For the blinking policy, since there is no preprocessing and nodes transition every blink interval τ, $T_{pre} = 0$ and $M = \frac{T*(N/k)}{\tau}$.

For the priority policy, the preprocessing time is proportional to the size of the input data and the network/disk I/O throughput, such that $T_{pre} = g(DataSize, I/O - throughput)$. The number of transitions then has two components. First, the minimum number of transitions assuming k of N nodes are always active, and second, any additional transitions that may occur due to variations in the power signal $P(t)$. Thus, $M = \sum_{i=0}^{\lceil \frac{n}{k-1} - 1 \rceil} (n - i(k-1)) + f(P(t))$, where $f(P(t))$ represents the number of transitions due to variations in $P(t)$.

Given the GreenSort running time above, we can derive energy-agility (in records per joule) by computing the energy E available over the running time based on $P(t)$ and then dividing the work done (in terms of the number of sorted records) by E. Thus, the overhead terms in the equations above increase the energy E in the denominator, thereby decreasing the energy-agility. Note that the model above is general and applies to any task that involves a synchronized all-to-all communication phase across all servers. By contrast, an embarrassingly-parallel task that requires no synchronization across servers, where each server must perform the same amount of work, would have a running time of only $T * (\frac{N}{k}) + d * N/k$. Here, the first term is the same as above, while the second term simply represents the transition latency incurred every time a set of k servers completes its work and activates another set of k servers.

4. IMPLEMENTATION

Our GreenSort implementation includes one worker process per server and a centralized power manager, both written in C++. The workers coordinate to sort the input data, while the power manager implements the power management policies from the previous section. We briefly discuss the worker and power manager implementations below, along with a description of our hardware platform.

Workers. We model our worker implementation after TritonSort [30], which divides the work of each sorting phase into a series of pipelined multi-threaded stages connected via producer-consumer buffers. However, since our focus is on energy-agility and not energy-efficiency, we do not optimize our workers for the highest possible efficiency on our platform. We also implement the necessary functions for workers to interact with the power manager to pause and resume its operation and report its progress with respect to other workers. Workers also include any functions necessary to support the various policies, such as pre-partitioning keys before distributing them with the priority policy.

Power Manager. The power manger monitors both server power usage and the amount of available power every τ, and then caps power by either altering servers' active power caps or activating/deactivating them. Deactivating one or more servers is a two-phase process. First, the power manager informs all workers of the servers that it is planning to deactivate. The workers on servers remaining active cleanly finish sending any outstanding buffers to the deactivating servers, while the workers on soon-to-be inactive servers cleanly finish sending all outstanding buffers to all other workers. Once finished, all workers send an acknowledgement to the power manager. The power manager subsequently deactivates servers by pausing their activity, and then remotely transitioning the server to an inactive power state. A similar two-phase process occurs when activating one or more servers. To implement our priority-based policies, the power manager also periodically polls each worker to track its progress with respect to other workers.

Platform. Our experimental platform is a set of Dell PowerEdge R720 servers, each with 32 2GHz cores, 64GB memory, and a 4TB disk. Since our platform combines a single local disk with 32 cores, it is not particularly energy-efficient for data-intensive applications, as many of its cores are largely idle during a sorting run. In particular, due the presence of only a single local disk, the servers are unable to maintain pure sequential I/O during key distribution, even when all servers are active, as the disk must concurrently read keys it sends and write keys it receives. As a result, we configure each worker to use a remote disk on a "dummy" server, as if it were another local disk, to ensure sequential I/O during key distribution. Of course, this paper's goal and the focus of our evaluation is not to construct the most efficient hardware platform, but to illustrate fundamental tradeoffs in energy-agile design.

Our servers include an external out-of-band management card that permits remote i) monitoring of server power usage every second, ii) power cycling, and iii) control of active power capping. The server's active power capping mechanism enables the power manager to set the cap between 85W (near the minimum idle power) and 285W (near the peak power). Unfortunately, the only inactive power state supported is Suspend-to-Disk (S4). The power manager transitions the server to S4 by executing a command line program, and transitions it out of S4 by remotely turning it on. The time to transition into and out of S4 has a lower bound of ~90s due to a required series of pre-boot tests.

Since our servers do not support Suspend-to-RAM (S3), which combines short power state transitions (on the order of seconds) with low power usage (~5% peak power), they impose a high overhead. S3 support is uncommon in servers; in fact, Dell makes no server that supports S3 and includes support for out-of-band management (a necessity in a remotely-managed datacenter). Due to these limitations, our power manager emulates other transition latencies by sleeping for a pre-determined amount of time when pausing and resuming servers. In addition, while we have access

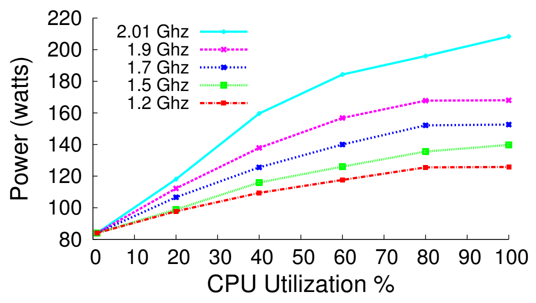


Figure 4: Power usage as a function of CPU utilization for each DVFS state on our servers.

to a shared cluster of 200 servers for experiments, we have only five dedicated servers that permit active power capping. Thus, in our evaluation, we use the cluster for experiments that focus on inactive power state transitions, and our dedicated servers for those that focus on active power capping.

5. EVALUATION

The goal of our evaluation is to use GreenSort to illustrate fundamental tradeoffs in the energy-agile design space. The design space is a function of many parameters, including the power signal characteristics, energy storage capacity, input data size, transition latency for inactive power states, etc. We first examine the limitations of using active power capping to satisfy power constraints to motivate the need for using inactive power states in our power management policies (§5.1). We then evaluate the use of inactive power states in each of our policy variants via microbenchmarks that alter the design parameters above in a controlled way to reveal their relative effect on performance (§5.2). Finally, we quantify the performance of each policy variant for real-world power traces on our platform (§5.3).

5.1 Limitations of Active Power Capping

Each of our servers permits setting an active power cap as low as 85W, which they enforce by throttling CPUs. Figure 4 shows the active power range of the servers' CPUs using DVFS, where the x-axis is the average CPU utilization across all cores (and where the network card and disk are idle). The graph shows that at 100% CPU utilization our servers' base power usage is 80W and their power usage in the highest DVFS state at 100% CPU utilization is near 215W, providing a 135W active power range for the CPU. The network interface card (NIC) and disk consume an additional 35W apiece when in use (at any utilization), resulting in an active power range (assuming any network and disk activity) of 150W to 285W. The server enforces active power caps below 150W by rapidly toggling CPUs between idle sleep modes (or C-states), which is similar, in principle, to blinking, although the C-state transitions are much faster (order of milliseconds or less).

Since our servers' CPU capacity is over-provisioned for data-intensive tasks like sorting, our workers only operate at 20% CPU utilization (averaged over all CPUs) in the lowest DVFS power state, which results in 95W CPU consumption and is already within ∼15W of the platform's minimum, idle power state. Thus, the total server power, when including the NIC and disk, during key distribution is 165W. In practice, our platform has little room to use DVFS to cap power, and must use C-state throttling. Figure 5 shows the limitations of active power capping. Here, the x-axis represents a constant power cap, while the y-axis is the running time to sort 125GB of data across five servers. In one case, we keep all servers active and cap power using our server's built-in active power capping mechanism by dividing the available power equally across the five servers, while in the other case, we satisfy the cap by using either the priority policy or the blinking policy (assuming a transition latency of 30 seconds).

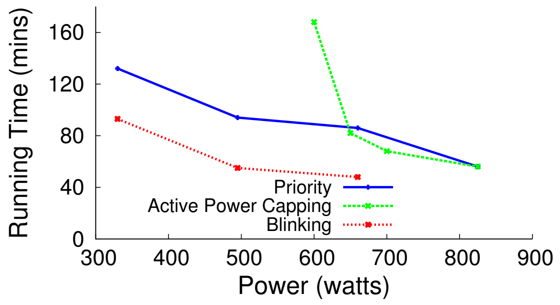


Figure 5: Sort run time for a constant power cap (on the x-axis) for different energy-agility policies.

The figure shows that once the cap drops below 660W (or 165W per server) the performance from only keeping three servers (or less) concurrently active outperforms keeping all servers active and actively capping power below 165W. For example, a 550W cap enables three servers to be concurrently active at 183W, or five servers to be concurrently at 110W. The former case completes in 95 minutes, while the latter takes more than 23 hours. This result highlights the importance of minimizing wasted power when using active power capping. Both the priority and blinking policies generally make better use of the available power. Note that we plot running time rather than energy-agility here only for ease of exposition: since average power is the same for each run, running time is proportional to energy-agility.

By concentrating power on fewer servers, the priority and blinking policies incur much less overhead than using active power caps across five servers. The overhead arises for at least three reasons. First, the more servers that are active, the more base power is wasted. For example, consider a scenario where there are three servers active, each with an active power cap of 130W, with 105W left to distribute. Activating another server with an active power cap of 105W is less efficient than increasing the cap of the active servers by 35W each, since the new active server will only use 25W for productive work (since 80W is the base power), while the three active servers will use all 105W for productive work. Second, the more servers that are active, the more CPU and power is devoted to additional OS and worker software overhead. Finally, adhering to low power caps not satisfiable using DVFS incurs increasing overhead due to frequently toggling CPUs into and out of idle C-state sleep modes. In our experiment, with an active power cap of 110W, almost no power goes to doing productive work.

Other platforms beyond our own are similarly constrained in using active power capping. If our platform were to have ten disks, rather than one, the non-energy-proportional disks would dominate power usage, reducing the effectiveness of active CPU power states. Similarly, our platform could use a low-power CPU, such as an Intel Atom, to operate at a higher utilization when sorting. However, as before, the

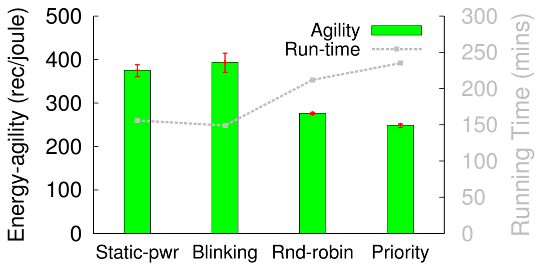


Figure 6: Baseline energy-agility (left y-axis) and running time (right y-axis) for each policy variant.

CPU's fraction of power would decrease relative to the non-energy-proportional disk and network card. While replacing mechanical disks with solid-state drives (SSDs) would lower the disks' fraction of power usage, low-power CPUs generally have many fewer active power states than high-power, multicore processors. For example, a low-power SuperMicro server in our lab, which has an Intel Atom processor, has no DVFS states and only a single C-state. Thus, while low-power servers may be energy-efficient, they often have a much narrower active power range than high-power servers.
Result: *Active power capping has limited benefits for I/O-intensive applications that do not fully utilize the CPU.*

5.2 Microbenchmarks

Given active power capping's limitations, we use microbenchmarks to quantify the design space of GreenSort's policies that use inactive power capping. Our baseline microbenchmark sorts 500GB across 10 servers, assuming a latency of 30 seconds to transition to an inactive power state, and a minimum energy storage capacity capable of supporting $\tau = 2$min. We use a power signal that oscillates between 25% and 75% peak power every τ, and, for the round-robin policy, we set the time-slice to 2 minutes. From our baseline benchmark, we then vary each parameter to quantify its relative impact on performance among the policies.

To set context, Figure 6 shows the results in our baseline scenario, where the left y-axis indicates the energy-agility and the right y-axis indicates the running time of the sorting system. For each policy variant, the energy-agility (in records sorted per joule of energy available) is indicated by a bar and the running time by a point. We execute each run three times and plot the minimum, maximum, and average energy-agility, which is within 1% or less across each separate run. The graph shows that the blinking policy outperforms all other policies both in terms of being energy-agile and overall running time in our baseline case. Also, the round-robin priority policy performs better than the strict priority policy. For both the priority and round-robin policies we use the pre-partitioning optimization. Without it, the 500GB sort takes more than 20 hours to complete less than 15% of key distribution, reflecting the two orders of magnitude performance decrease from using purely random, rather than sequential, I/O. However, despite the optimization, the additional I/O required by the priority and round-robin policies in this case outweighs any additional transition overhead from the blinking policy.
Transition Latency. Figure 7 shows the performance of our three policies as the transition latency varies. The graph illustrates that the blinking policy outperforms the priority-

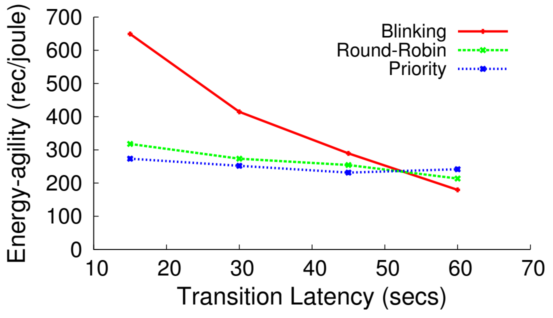


Figure 7: Transition latency's effect on agility.

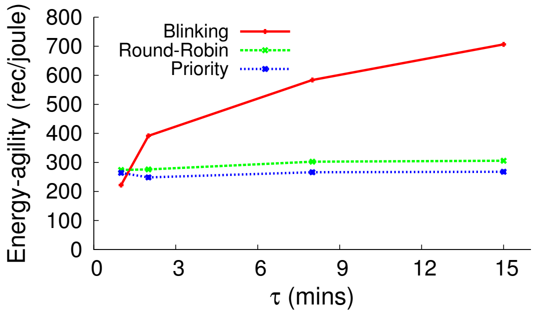


Figure 8: Power signal frequency's effect on agility (represented by energy storage capacity τ).

based policies by as much as 2X for short latencies. Even for long latencies of 45 seconds, blinking's performance remains better than priority despite the fact that blinking incurs the transition latency overhead once every $\tau = 2$ minutes. For example, with a latency of 45 seconds, the blinking policy spends 37.5% of its time simply transitioning power states. However, once the transition latency exceeds one minute, the priority policy outperforms blinking, as the overhead of transitioning begins to outweigh the benefit of blinking's fewer number of I/Os. As expected, the performance of the round-robin priority policy falls between the blinking and priority policies regardless of the latency.
Power Signal and Energy Storage. The frequency of variation in the power signal also affects performance. One way to alter the frequency is by changing the amount of minimum energy storage capacity, as represented by the length of τ. Figure 8 shows that the blinking policy's performance improves as energy storage capacity (and τ) increases from one to 15 minutes, since it reduces the frequency of blinking and its associated overhead. In contrast, energy storage does not significantly affect the priority or the round-robin policy, since they do not transition every interval τ.

We also alter the variability of the power signal without changing τ, as depicted in Figure 9. Here, the x-axis represents the length of each period of constant power, such that the power signal changes every x minutes; the higher x, the less variable the power signal. The graph shows the relative performance of the priority policy improves as the variability decreases (and the periods of constant power increase), since it reduces the impact of its imbalanced progress. In contrast, the power signal's variability does not improve the performance of blinking, since it maintains balanced progress and its transition overhead is independent of the variability.
Job Size. The length of a sorting run also affects the relative performance of the policies. While a longer running

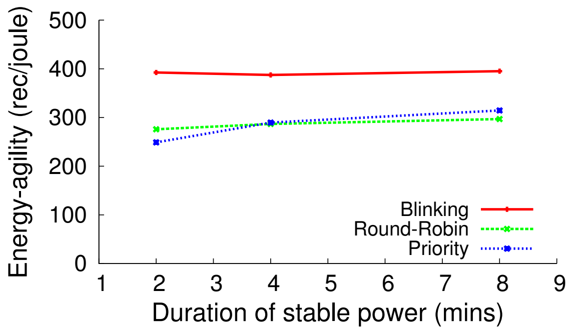


Figure 9: Power variability's effect on agility.

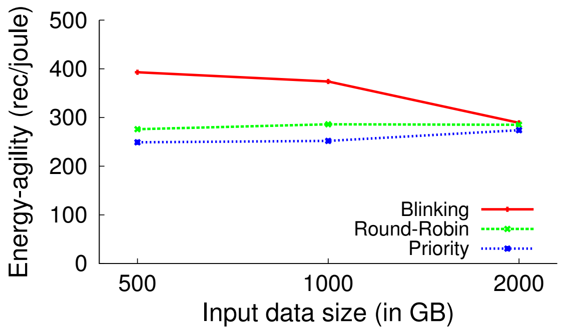


Figure 10: Input datasize's effect on agility.

time causes more transition overhead for the blinking policy, if the increase is due to fewer resources or a larger input data size, it also causes more additional I/O overhead for the priority policy. Figure 10 shows the results of increasing the input data size to sort in our baseline microbenchmark. The graph demonstrates that the larger the input data size the worse blinking (and round-robin) performs relative to the priority policy. For example, when sorting 500GB, blinking is 58% more agile, when sorting 1TB it is 48% more agile, and when sorting 2TB it is only 5% more agile. The worsening relative performance reflects the fact that blinking's transition overhead is a function of the running time, and not the variability of the power signal, as with the priority policy. Thus, as the running time becomes longer the transition overhead increasingly outweighs the additional I/O overhead due to the larger input data.

Result: *Energy-agile design is influenced by a variety of parameters, including a platform's transition latency, power signal characteristics, energy storage capacity, and job size.*

5.3 Real Power Signals

Finally, we evaluate our GreenSort policies on real power signals on our hardware platform to get a sense of performance for each policy in practice. Figure 11 shows the performance of each policy variant on the solar energy signal from Figure 1(b) and the electricity price signal from Figure 1(c). In the latter case, we set a fixed budget of 0.94¢ every five minutes for electricity (determined by the energy needed to power on five nodes at an average price of 3.27¢/kWh from our sample), which transforms the electricity price signal into a hard power cap. For comparison, we scale the solar and price traces such that they yield the same average power. In addition, we also compare our results with a uniformly random power signal and a static power signal equal to the average power of the solar and price signals. These experiments sort 1TB of data across 10 nodes.

In contrast to our microbenchmarks, the priority policy significantly outperforms the blinking policy (by more than 40% in each case). Since our actual platform has a transition latency of 90s, it restricts the blinking policy to utilizing the server for a maximum of 30s for each τ=120s cycle. As expected, the static power signal performs best since it makes maximum use of the priority policy to minimize transitions. We can compare each variable signal with this static signal to get a sense of how variations ultimately impact the running time and the energy-agility of the sorting system.

We define the energy-agility factor as the ratio of energy required to finish a sort under static power to the energy required to finish the same sort under a variable power, whose average is same as that of the static power. For example, the uniformly random signal yields an energy-agility factor of 0.37 for the blinking policy, 0.53 for the round-robin policy, and 0.65 for the priority policy. That is, sorting under a constant power consumes only 37% of the energy provided to the blinking policy under a varying power, or 65% of the energy provided to the priority policy. The trend is slightly worse for the blinking policy under the solar and spot price signals, since these signals are not uniformly random but have correlated periods of extended inactivity. The priority policy is 2X more agile than the blinking policy both under solar and spot price signals. We see that despite exceeding the two-read-two-write limit of distributed sorting, the priority and round-robin policies fare better than the blinking policy owing to the high transition overhead of our platform. In terms of energy agility factor, for the solar and price signals, respectively, the blinking policy yields 0.35 and 0.32, the priority policy yields 0.78 and 0.69, and the round-robin policy yields 0.71 and 0.67.

Result: *Since variations in power increase the time and energy to complete a task, energy-agile design is important when considering the benefits of using renewable energy sources or participating in demand response programs. As one example from above, sorting under a stable power will only consume 69% of the energy required by the best GreenSort policy (priority in this case), when the power varies based on real-time electricity prices. Thus, in this case, real-time prices should be at least 31% lower than fixed prices to warrant opting into using them.*

6. RELATED WORK

Energy-agility is related to energy-proportionality [6] in that it also benefits from energy-proportional servers capable of precisely varying their power usage over a wide active power range. However, a perfectly energy-proportional server would not necessarily optimize energy-agility, as energy-proportionality only dictates that server power usage increases linearly with utilization, regardless of the energy-efficiency at a particular utilization level. In contrast, energy-agility incorporates both energy-efficiency and the ability to rapidly adapt power usage over a wide dynamic power range. In addition, unlike energy-proportionality, energy-agility is power-driven, rather than workload-driven. While energy-proportionality applies directly to web applications and batch schedulers, where the workload intensity varies over time based on user request volume, it is not applicable to long-running parallel tasks, such as distributed sorting, with no variance in the workload. Thus, recent work on energy-efficiency focuses on designing balanced systems

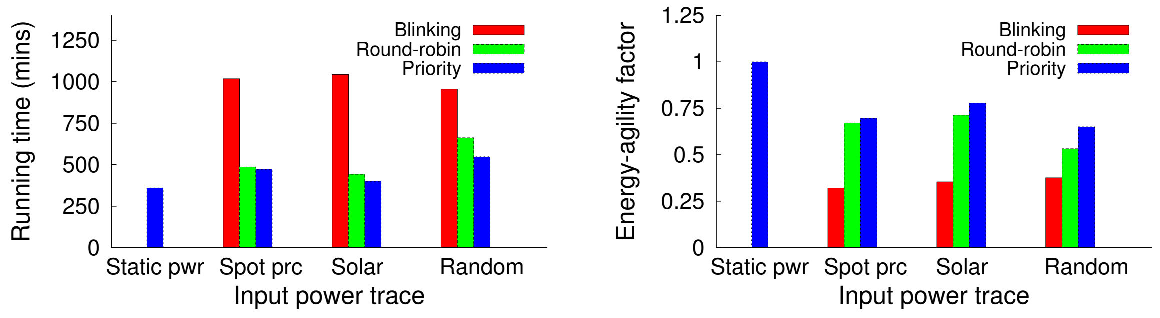


Figure 11: Performance on real power signals showing the effect on (a) running time and (b) agility.

that optimize peak performance per watt, e.g., at 100% utilization [4, 30, 31]. In contrast, energy-agility encourages applications to operate efficiently at all utilization levels.

Prior work on enforcing soft power caps generally focuses on time-shifting data center energy usage via energy storage [18, 39], workload scheduling [16, 17], or both [15] to reduce energy costs. This group of work exploits temporal and spatial variations in electricity prices to minimize costs without violating applications' quality-of-service (QoS) requirements, e.g., job deadlines or response latency requirements. In contrast, our work is not cost-oriented, but instead introduces energy-agility as a cost-independent metric to quantify an application's ability to adapt to variable power. Optimizing energy-agility is important for long-running delay-tolerant tasks, since adapting their power usage to available power is less expensive than masking power variations with energy storage [18, 39]. Long-running tasks are also less amenable to scheduling policies than short tasks, which may simply be deferred until power is cheap or plentiful [16, 17].

There has been much less prior work on adapting systems to dynamically-changing hard power caps. While a variety of power capping mechanisms exist for individual servers, these mechanisms ignore the inter-node dependencies that affect performance in distributed applications [7, 14, 22]. Thus, prior work focuses on regulating power using simplistic workloads, e.g., compute-intensive batch jobs with few data dependencies, that readily permit time-shifting workload to satisfy power caps [24]. Finally, the only prior work we are aware of that accounts for inter-node dependencies when capping power focuses on interactive services, e.g., a distributed memory cache [32] and file system [33], and not the delay-tolerant jobs that are most amenable to demand-side management. However, we show that a similar blinking abstraction applies to these workloads, albeit differently than with interactive workloads.

7. CONCLUSION

This paper introduces energy-agility as a metric to evaluate green datacenter applications that adapt to power variations, and then design GreenSort to illustrate fundamental tradeoffs in energy-agile design. While we focus on sorting, we believe our experience in designing GreenSort reveals some general lessons for energy-agile design that are applicable to a broader range of data-intensive applications, such as MapReduce [8]. We summarize these lessons below.

Inactive Power Capping is Useful, Despite its Overhead. Prior work largely focuses on active power capping [10, 25, 27, 28], since inactive power capping is not appropriate for all workloads. For example, online data-intensive (OLDI) workloads may need immediate access to data stored on any server at any time, and, thus, cannot incur the transition latency associated with inactive power capping [25, 27]. In contrast, for data-intensive, parallel batch jobs, such as sorting, we show that inactive power capping is much more efficient than using active power capping because it concentrates more power on doing useful work. This useful work offsets the transition overhead associated with inactive power capping. In addition, while applications could employ active power capping to satisfy caps as low as ~50% peak power, we find setting active caps, which slow down server progress to activate additional servers, is not beneficial due to high server idle power.

Blinking is Preferred when Coordination is Necessary. While blinking incurs high latencies at regular short intervals, it does not affect an application's pattern of remote I/O, since servers are always concurrently active for some fraction of each interval. In contrast, any policy that deactivates some fraction of servers has the potential to alter applications' remote I/O patterns and degrade performance, e.g., by changing sequential I/O to random I/O, since servers may not always be concurrently active. In addition to sorting, MapReduce and other "big data" platforms also have frequent periods of large-scale coordinated data movement. In contrast, the priority policy works well for embarrassingly parallel tasks that require no coordination, since it minimizes transition overheads.

When Deactivating Servers, Organizing Data is Beneficial. As the transition latency increases, the useful work performed when blinking decreases. At some point, minimizing this overhead by deactivating servers, as per our priority policy, becomes attractive. Since naïvely deactivating servers affects I/O patterns, actively organizing the data in conjunction with the power management policy is important. As we show, incurring additional I/O upfront to maximize sequential I/O later can improve performance. Since today's server platforms do not support ACPI's S3 state, their transition latencies warrant this approach.

Our results suggest that energy-agile design is a potentially rich area for future research, especially given the diminishing returns on improving energy-efficiency and the increasing use of variable power. Our work shows the importance of energy-agility in quantifying how power variations increase the time and energy to complete a task.

Acknowledgements. This research is funded by the National Science Foundation under grants 1422245, 1405826, and 1339839.

8. REFERENCES

[1] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *NSDI*, April 2009.

[2] A. Aggarwal and J. Vitter. The Input/Output Complexity of Sorting and Related Problems. 1988.

[3] B. Aksanli, J. Venkatesh, L. Zhang, and T. Rosing. Utilizing Green Energy Prediction to Schedule Mixed Batch and Service Jobs in Data Centers. In *HotPower*, October 2011.

[4] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, October 2009.

[5] Apple and the Environment. http://www.apple.com/environment/renewable-energy/, Accessed November 2013.

[6] L. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12), December 2007.

[7] R. Cochran, C. Hankendi, A. Coskun, and S. Reda. Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In *MICRO*, December 2011.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, December 2004.

[9] Q. Deng, D. Meisner, L. Ramos, T. Wenisch, and R. Bianchini. MemScale: Active Low-Power Modes for Main Memory. In *ASPLOS*, March 2011.

[10] X. Fan, W. Weber, and L. Barroso. Power Provisioning for a Warehouse-sized Computer. In *ISCA*, June 2007.

[11] State of the Markets Report 2008. Technical report, Federal Energy Regulatory Commission, August 2009.

[12] K. Fehrenbacher. The Era of the 100MW Data Center. In *Gigaom*, January 31 2012.

[13] K. Finley. Facebook Says its New Data Center Will Run Entirely on Wind. In *Wired*, November 13th 2013.

[14] A. Gandhi, M. Harchol-Balter, R. Das, J. Kephart, and C. Lefurgy. Power Capping via Forced Idleness. In *Weed*, June 2009.

[15] I. Goiri, W. Katsak, K. Le, T. Nguyen, and R. Bianchini. Parasol and GreenSwitch: Managing Datacenters Powered by Renewable Energy. In *ASPLOS*, March 2013.

[16] I. Goiri, K. Le, M. Haque, R. Beauchea, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenSlot: Scheduling Energy Consumption in Green Datacenters. In *SC*, April 2011.

[17] I. Goiri, K. Le, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenHadoop: Leveraging Green Energy in Data-Processing Frameworks. In *EuroSys*, 2012.

[18] S. Govindan, A. Sivasubramaniam, and B. Urgaonkar. Benefits and Limitations of Tapping into Stored Energy for Datacenters. In *ISCA*, June 2011.

[19] D. Irwin, N. Sharma, and P. Shenoy. Towards Continuous Policy-driven Demand Respone in Data Centers. In *GreenNets*, August 2011.

[20] J. Koomey. Growth in Data Center Electricity Use 2005 to 2010. In *Analytics Press*, Oakland, California, August 2011.

[21] A. Krioukov, C. Goebel, S. Alspaugh, Y. Chen, D. Culler, and R. Katz. Integrating Renewable Energy Using Data Analytics Systems: Challenges and Opportunities. In *Bulletin of the IEEE Computer Society Technical Committee*, March 2011.

[22] C. Lefurgy, X. Wang, and M. Ware. Server-level Power Control. In *ICAC*, February 2007.

[23] C. Li, A. Qouneh, and T. Li. iSwitch: Coordinating and Optimizing Renewable Energy Powered Server Clusters. In *ISCA*, June 2012.

[24] Z. Liu, A. Wierman, Y. Chen, B. Razon, and N. Chen. Data Center Demand Response: Avoiding the Coincident Peak via Workload Shifting and Local Generation. 70(10), 2013.

[25] D. Lo, L. Cheng, R. Govindaraju, L. Barroso, and C. Kozyrakis. Towards Energy Proportionality for Large-scale Latency-Critical Workloads. In *ISCA*, June 2014.

[26] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. In *ASPLOS*, March 2009.

[27] D. Meisner, C. Sadler, L. Barroso, W. Weber, and T. Wenisch. Power Management of On-line Data Intensive Services. In *ISCA*, June 2011.

[28] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers. In *ISCA*, June 2006.

[29] A. Rasmussen, M. Conley, R. Kapoor, V. Lam, G. Porter, and A. Vahdat. Themis: An I/O-Efficient MapReduce. In *SoCC*, October 2012.

[30] A. Rasmussen, G. Porter, M. Conley, H. Madhyasthay, R. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced and Energy-Efficient Large-Scale Sorting System. *TOCS*, 31(1), February 2013.

[31] S. Rivoire, M. Shah, and P. Ranganathan. JouleSort: A Balanced Energy-Efficient Benchmark. In *SIGMOD*, June 2007.

[32] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: Managing Server Clusters on Intermittent Power. In *ASPLOS*, March 2011.

[33] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. A Distributed File System for Intermittent Power. In *IGCC*, June 2013.

[34] R. Singh, D. Irwin, P. Shenoy, and K. Ramakrishnan. Yank: Enabling Green Data Centers to Pull the Plug. In *NSDI*, April 2013.

[35] Sort Benchmark Home Page. http://sortbenchmark.org/, Accessed July 2014.

[36] C. Stewart and K. Shen. Some Joules Are More Precious Than Others: Managing Renewable Energy in the Datacenter. In *HotPower*, October 2009.

[37] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu. Delivering Energy Proportionality with Non-Energy-Proportional Systems: Optimizing the Ensemble. In *HotPower*, December 2008.

[38] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the Energy Efficiency of a Database Server. In *SIGMOD*, June 2010.

[39] R. Urgaonkar, B. Urgaonkar, M. Neely, and A. Sivasubramaniam. Optimal Power Cost Management Using Stored Energy in Data Centers. In *SIGMETRICS*, March 2011.

[40] A. Wierman, Z. Liu, I. Liu, and H. Mohsenian-Rad. Opportunities and Challenges for Data Center Demand Response. In *IGCC*, June 2014.